PRINCETON UNIVERSITY

iris hep

Awkward Array

Peter Fackeldey

6/6/25

About me

Postdoctoral Research Associate at Princeton in the IRIS-HEP Project

Scientific background in High Energy Physics (HEP):

- Member of the CMS collaboration (2nd largest collaboration at CERN) since 9 years (ATLAS & CMS found the Higgs boson in 2012)
- PhD thesis about Higgs boson pair production with the CMS experiment at CERN

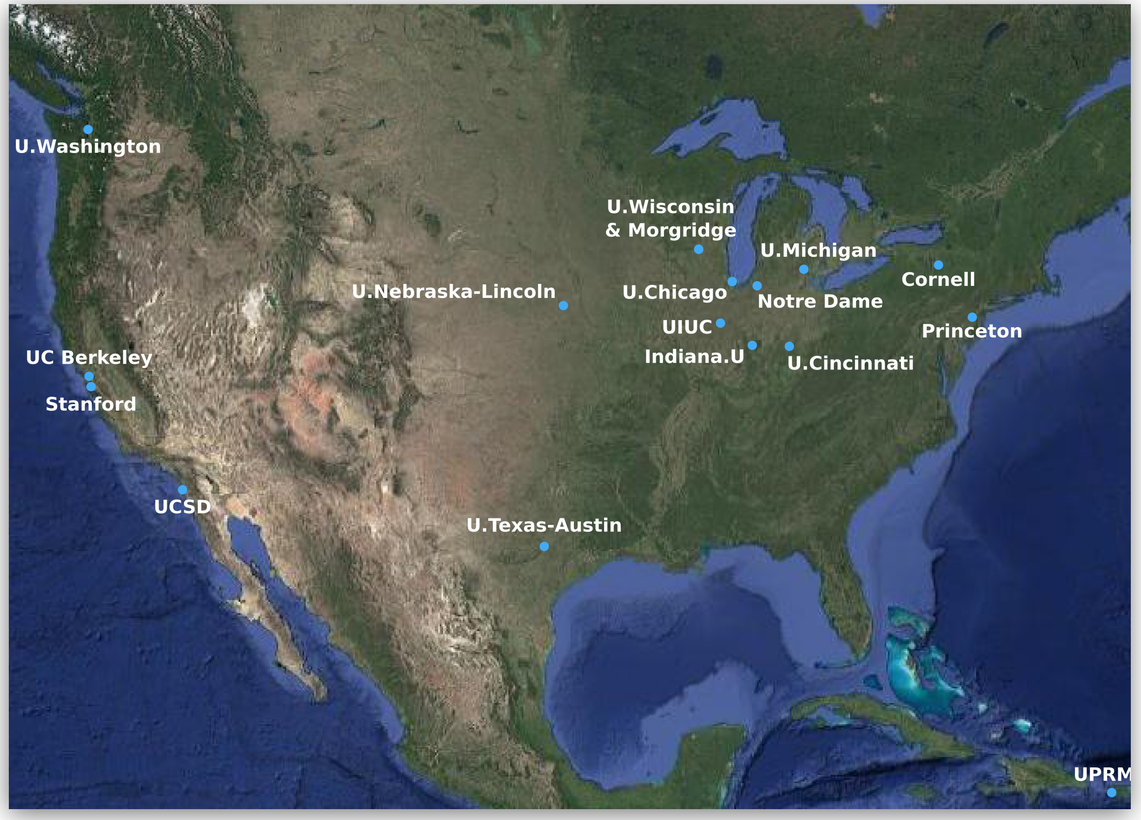
Software background:

- Python is my main programming language since ~10 years
- I moved from being a “power-user” and contributor of the Scikit-HEP domain stack to being a maintainer
- I have a passion for “number-crunching” Python software (talk to me about JAX!)

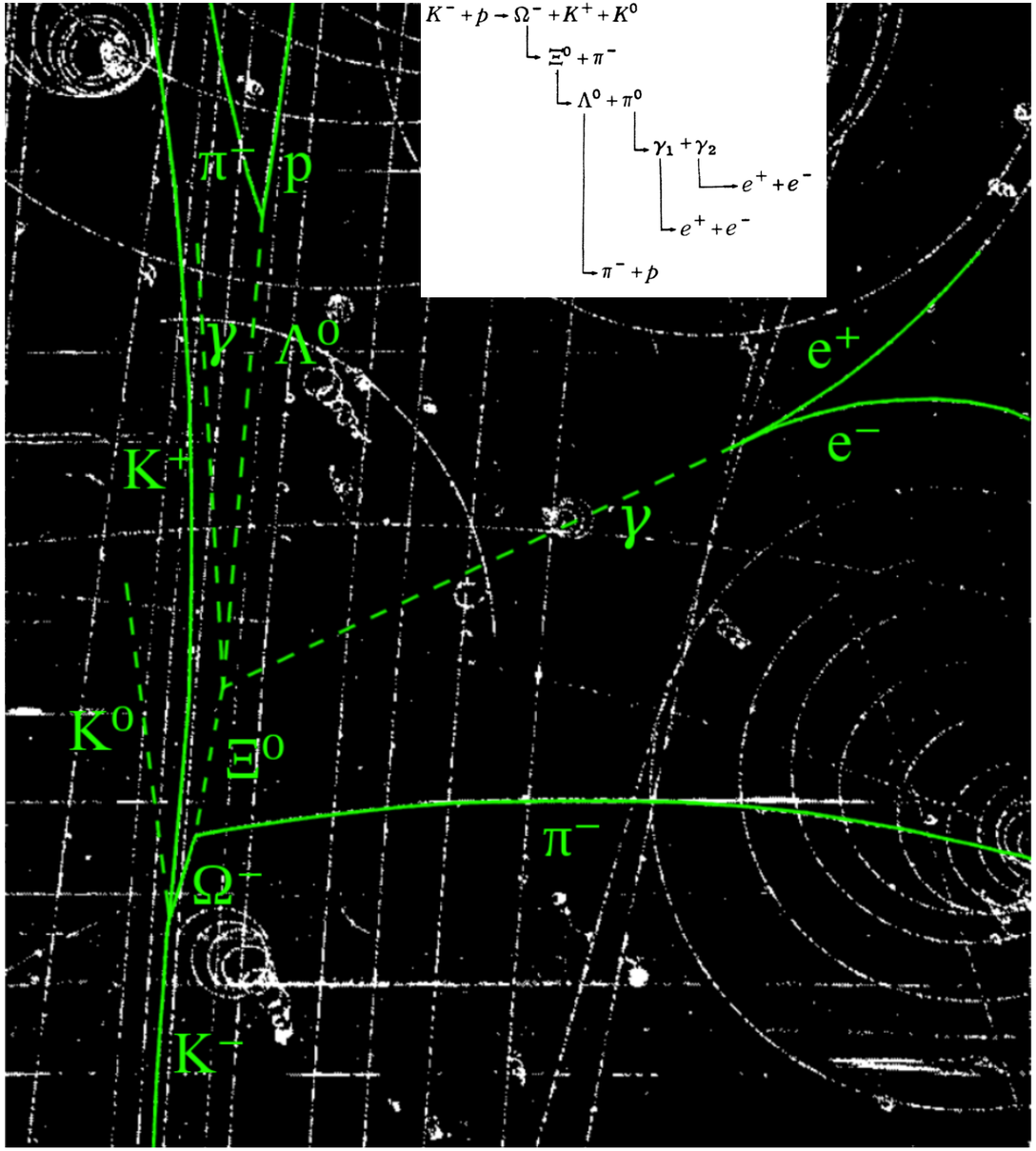
About us (IRIS-HEP & Scikit-HEP)

Scikit-HEP is our community project/domain stack on Github (https://github.com/scikit-hep)

IRIS-HEP is a software institute funded by the NSF (https://iris-hep.org/)



High Energy Physics: introduction



Understanding the fundamental building blocks of our universe is *a lot* about particle interactions

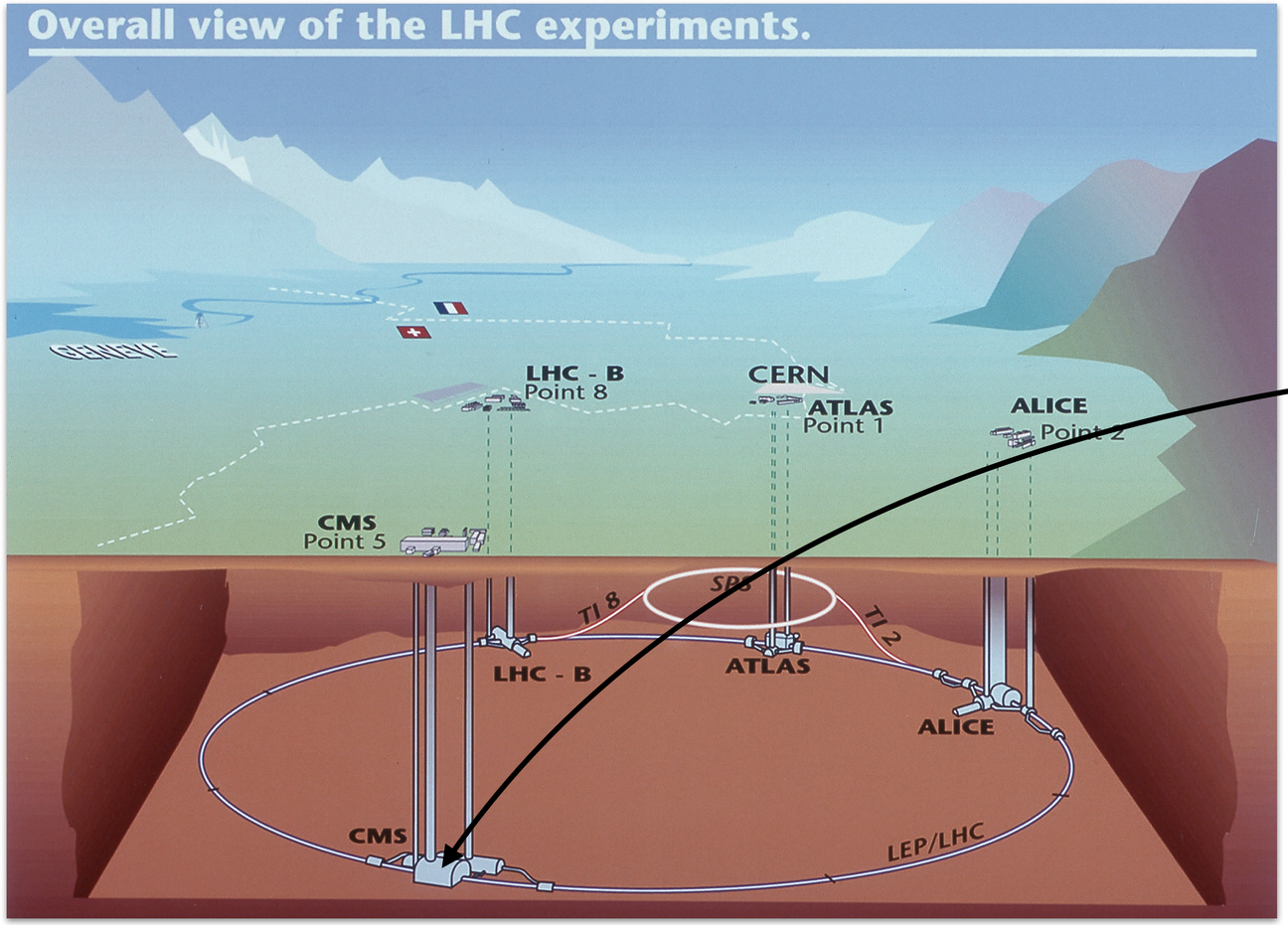
→ shoot particles against each other & take pictures to probe interactions

Analysis of these pictures:

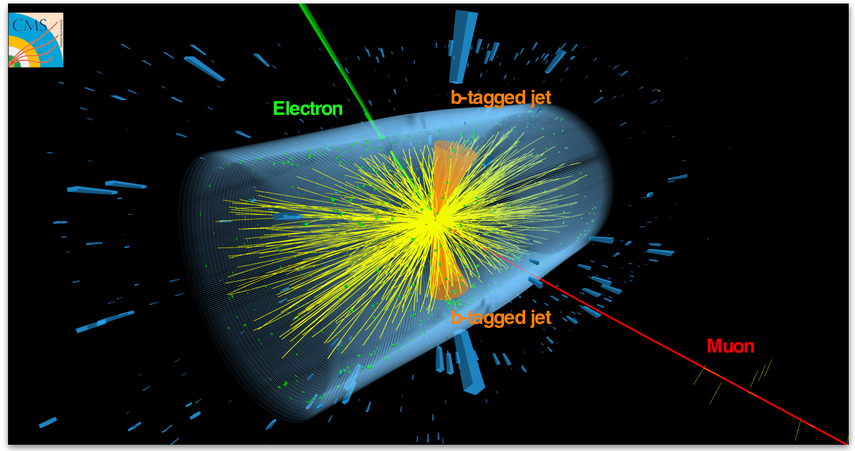
1. **Isolate** interesting tracks (lines)
2. **Combinatorics**: e.g. find parent particles
3. **Reconstruct** physical properties (mass)

In the ‘60s they manually analyzed ~100.000 pictures to find this one

High Energy Physics: nowadays



Is this new physics?

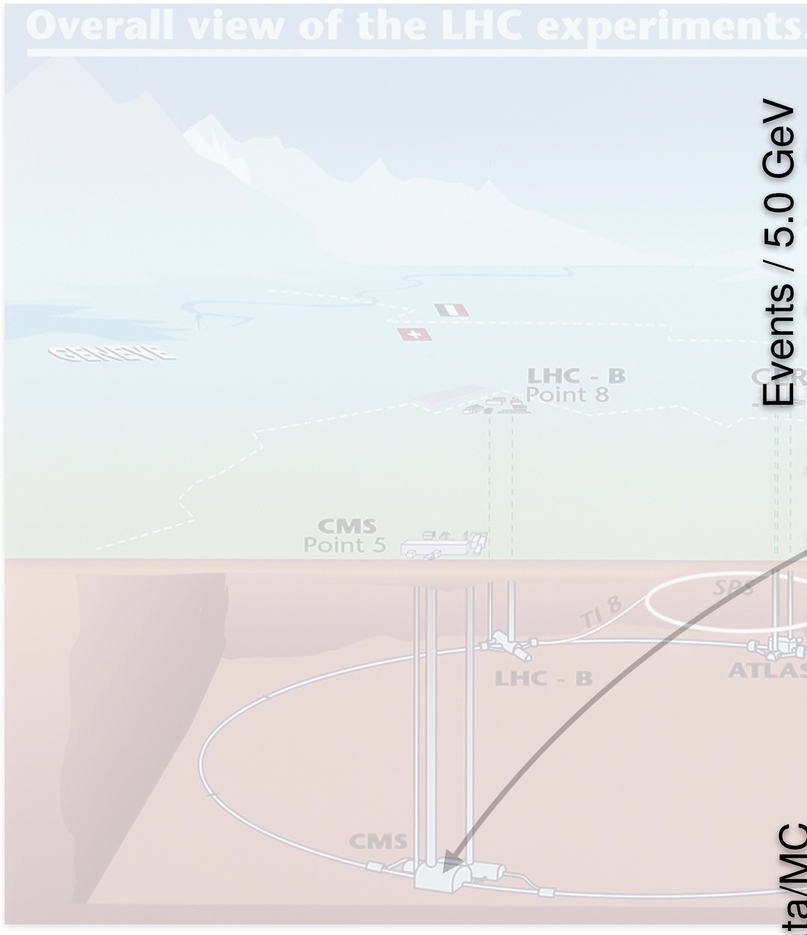


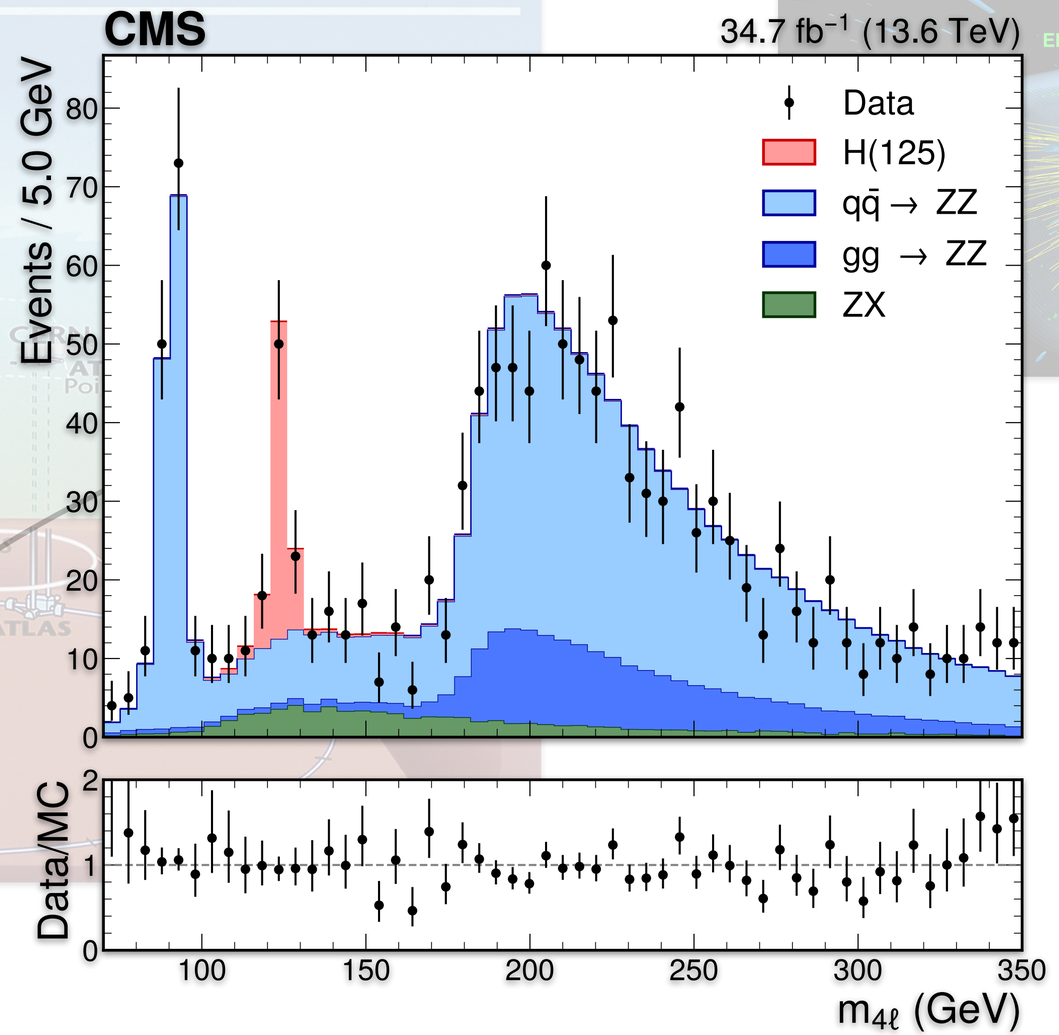
Analysis of these events (same as before):

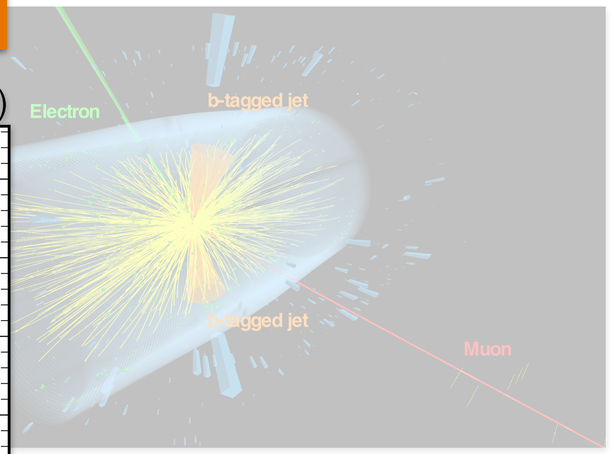
1. **Isolate** interesting tracks (lines)
2. **Combinatorics**: e.g. find parent particles
3. **Reconstruct** physical properties (mass)

High Energy Physics: nowadays

If you're really good at these steps
you might find a Higgs boson 🎉







High Energy Physics: computational view

- Large amounts of data
  - 40 million events (“collisions”) per second
  - Each event is ~1MB of data → 40TB/s data rate

Distributed computing

- Every event is independent

Vectorization opportunity

- Every event produces a *variable* number of particles

Jagged data

- Detectors measure a collection of information *per particle*: momentum, energy, position, …

Arrays of structs

- Combining decay products into parent particles

JOIN (in SQL)

High Energy Physics: computational view

- Large amounts of data
  - 40 million events (“collisions”) per second
  - Each event is ~1MB of data → 40TB/s data rate

Distributed computing

- Every event is independent — Vectorization opportunity
- Every event produces a *variable* number of particles — Jagged data
- Detectors measure a collection of information *per particle*: momentum, energy, position, … — Arrays of structs
- Combining decay products into parent particles — JOIN (in SQL)

Awkward Array: *represent* & *manipulate* jagged data efficiently

Awkward Array

Awkward Array

pypi package 2.8.1 | conda-forge v2.8.1 | python 3.9–3.13 | license BSD 3-Clause | Tests passing

Scikit-HEP Project | NSF 1836650 | DOI 10.5281/zenodo.4341376 | docs online | chat online

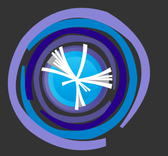
Starred 874

Awkward Array is a library for **nested, variable-sized data**, including arbitrary-length lists, records, mixed types, and missing data, using **NumPy-like idioms**.

Arrays are **dynamically typed**, but operations on them are **compiled and fast**. Their behavior coincides with NumPy when array dimensions are regular and generalizes when they're not.

Developed (*mainly*) by Princeton (PICSciE & Physics):
Jim Pivarski*, Angus Hollands*, Ianna Osborne, Henry Schreiner, Andres Rios-Tascon, Peter Fackeldey
(*changed positions)

Awkward Array: Preview



```
array = ak.Array([
    [{"x": 1.1, "y": [1]}, {"x": 2.2, "y": [1, 2]}, {"x": 3.3, "y": [1, 2, 3]}],
    [],
    [{"x": 4.4, "y": [1, 2, 3, 4]}, {"x": 5.5, "y": [1, 2, 3, 4, 5]}]
])
```

NumPy-like expression

```
output = np.square(array["y", ..., 1:])
```

```
[
    [[], [4], [4, 9]],
    [],
    [[4, 9, 16], [4, 9, 16, 25]]
]
```

4.6 seconds to run (2 GB footprint)

equivalent Python

```
output = []
for sublist in python_objects:
    tmp1 = []
    for record in sublist:
        tmp2 = []
        for number in record["y"][1:]:
            tmp2.append(np.square(number))
        tmp1.append(tmp2)
    output.append(tmp1)
```

138 seconds to run (22 GB footprint)

(single-threaded on a 2.2 GHz processor with a dataset 10 million times larger than the one shown)

Credit: Jim Pivarski

Jagged Arrays: basics (1/2)

Representation:

User code:

```
energy = ak.Array([
  [50, 100, 30],          # event 1
  [5],                    # event 2
  [40, 20],               # event 3
])
```

Memory layout:

```
energy-starts = 0                3   4
energy-stops = 3                 4   6

energy-data   = 50, 100, 30, 5, 40, 20
```

Jagged Arrays: basics (1/2)

Representation:

User code:

```
energy = ak.Array([
  [50, 100, 30],          # event 1
  [5],                    # event 2
  [40, 20],               # event 3
])
```

Memory layout:

```
energy-starts = 0          3   4
energy-stops = 3           4   6

energy-data   = 50, 100, 30, 5, 40, 20
```

Manipulation (“remove first energy from all events”):

User code:

```
energy[:, 1:]
>> ak.Array([
  [100, 30],              # event 1
  [ ],                    # event 2
  [20],                   # event 3
])
```

Memory layout:

```
energy-starts = 1          4  5     ←+1
energy-stops = 3           4  6

energy-data   = 50, 100, 30, 5, 40, 20
```

Jagged Arrays: basics (2/2)

Manipulation (“square root of each energy”):

User code:

```
np.sqrt(energy)
>> ak.Array([
  [7.07, 10, 5.48],      # event 1
  [2.24],                # event 2
  [6.32, 4.47],          # event 3
])
```

Memory layout:

energy-starts = 0 3 4

energy-stops = 3 4 6

$\sqrt{\text{energy-data}}$ = 7.07, 10, …, 6.32, 4.47

Manipulation (“sum all energies in an event”):

User code:

```
ak.sum(energy, 1)
>> ak.Array([
  50+100+30=180,      # event 1
  5=5,                # event 2
  40+20=60,           # event 3
])
```

Memory layout:

~~energy-starts~~

~~energy-stops~~

that’s a flat array!

$\sum$ energy-data per collision = 180, 5, 60

14 Jagged Arrays: types of manipulations

Structural manipulations:

"remove first energy from all events"

→ Needs only starts & stops (or derivates, e.g. "offsets")

→ Very efficient: doesn't need data (allows for IO optimizations)

array**[:, :1]**

Data manipulations:

"square root of each energy"

→ Independent of structure (only needs flat contents/data)

→ Route fast NumPy kernels to flat data array

np.sqrt(array**)**

Complex jagged manipulations:

"sum all energies in an event"

→ Needs structural information (starts & stops) *and* data

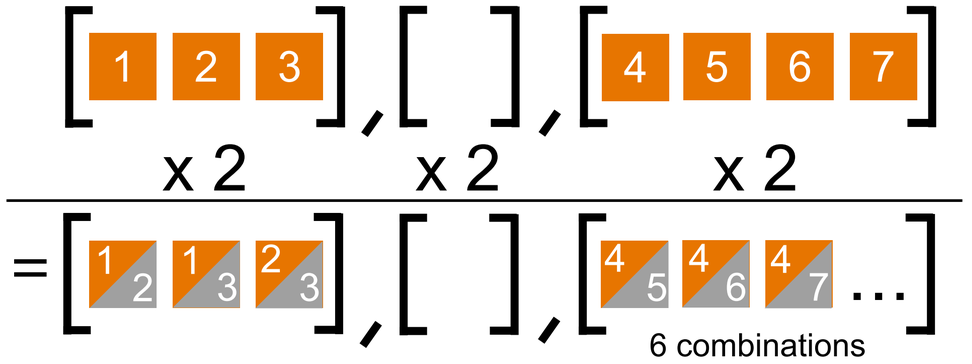
→ Dedicated kernels (awkward-cpp)

ak.sum(array, **1)**

15 Jagged Arrays: combinatorics

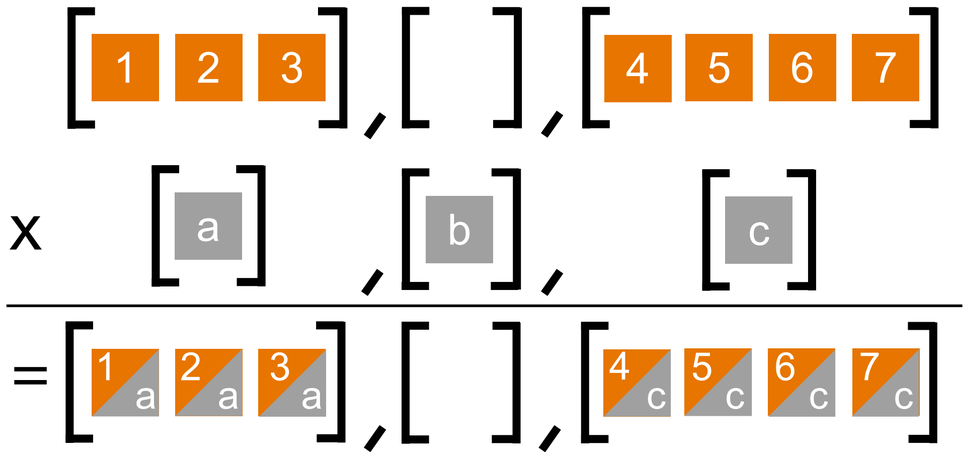
Combinations (“SELF JOIN”):

```
In [  ]: x = ak.Array([[1, 2, 3], [], [4, 5, 6, 7]])

In [  ]: ak.combinations(x, 2).show()
[[(1, 2), (1, 3), (2, 3)],
 [],
 [(4, 5), (4, 6), (4, 7), (5, 6), (5, 7), (6, 7)]]
```



Cartesian product (“CROSS JOIN”):

```
In [  ]: x = ak.Array([[1, 2, 3], [], [4, 5, 6, 7]])

In [  ]: y = ak.Array([["a"], ["b"], ["c"]])

In [  ]: ak.cartesian([x, y]).show()
[[(1, 'a'), (2, 'a'), (3, 'a')],
 [],
 [(4, 'c'), (5, 'c'), (6, 'c'), (7, 'c')]]
```



RecordArrays: semantically group arrays

RecordArrays group arrays into a meaningful structure:

events.Electron.mass

Particle type

Physical property

"*events* is a RecordArray of *particle types*, where each type is a RecordArray of different *physical properties*"

Fields are of fixed length, typed, and (possibly) named

RecordArrays: Example of HEP Events

Event 1

Electron 1
p_T = 45 GeV
η = 1.1
ϕ = 0.1

Electron 2
p_T = 23 GeV
η = 0.2
ϕ = 1.2

Muon 1
p_T = 45 GeV
η = 1.1
ϕ = 0.1

Muon 2
p_T = 23 GeV
η = 0.2
ϕ = 1.2

Muon 3
p_T = 23 GeV
η = 0.2
ϕ = 1.2

Event 2

Electron 1
p_T = 43 GeV
η = 0.0
ϕ = 0.2

Electron 2
p_T = 76 GeV
η = 0.4
ϕ = 2.1

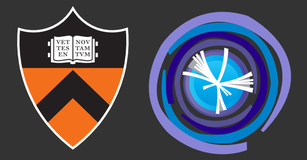
Muon 1
p_T = 32 GeV
η = 1.15
ϕ = 0.9

Photon 1
p_T = 43 GeV
η = 0.3
ϕ = 0.4
ID = 0.87

Photon 2
p_T = 76 GeV
η = 0.8
ϕ = 2.3
ID = 0.96

Each event may have **different number of particles with different physical properties**

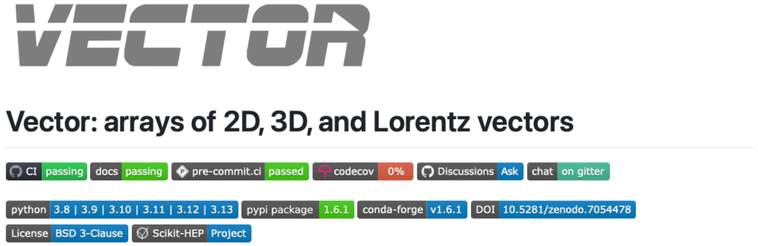
Reconstruction of physical properties



- Awkward Array allows to add behaviors to RecordArrays:

  If an array has the *named* fields “energy” and “momentum”, we can automatically provide a way to calculate the mass

- All of these physics quantities are abstracted into behaviors via *vector*
  (a package to work with Lorentz vectors)



- For more complex reconstructions Awkward Array provides:
  - Interoperability with JIT-compiled for-loop expressions in Numba, Julia (Awkward.jl), and C++ (cppyy)
  - Various conversions to ML libraries (PyTorch, TensorFlow, JAX)

19 JIT compilation with Numba

```
In [  ]: import awkward as ak; import numpy as np; import numba as nb

In [  ]: array = ak.Array([
    ...:     [{"x": 1.1, "y": [1]}, {"x": 2.2, "y": [1, 2]}, {"x": 3.3, "y": [1, 2, 3]}], # sum'y': 10
    ...:     [],                                                                           # sum'y': 0
    ...:     [{"x": 4.4, "y": [1, 2, 3, 4]}, {"x": 5.5, "y": [1, 2, 3, 4, 5]}],           # sum'y': 25
    ...:     [{"x": 6.6, "y": [1, 2, 3, 4, 5, 6]}],                                       # sum'y': 21
    ...: ])

In [  ]: ak.sum(ak.sum(array.y, axis=-1), axis=-1)
Out[  ]: <Array [10, 0, 25, 21] type='4 * int64'>
```

20 JIT compilation with Numba

```
In [  ]: import awkward as ak; import numpy as np; import numba as nb

In [  ]: array = ak.Array([
    ...:     [{"x": 1.1, "y": [1]}, {"x": 2.2, "y": [1, 2]}, {"x": 3.3, "y": [1, 2, 3]}], # sum'y': 10
    ...:     [],                                                                           # sum'y': 0
    ...:     [{"x": 4.4, "y": [1, 2, 3, 4]}, {"x": 5.5, "y": [1, 2, 3, 4, 5]}],           # sum'y': 25
    ...:     [{"x": 6.6, "y": [1, 2, 3, 4, 5, 6]}],                                       # sum'y': 21
    ...: ])

In [  ]: ak.sum(ak.sum(array.y, axis=-1), axis=-1)
Out[  ]: <Array [10, 0, 25, 21] type='4 * int64'>

In [  ]: @nb.jit
    ...: def sum_of_y(array):
    ...:     out = np.zeros(len(array), dtype=np.int64)
    ...:     for i, list_of_records in enumerate(array):
    ...:         for record in list_of_records:
    ...:             for y in record.y:
    ...:                 out[i] += y
    ...:     return out
    ...:

In [  ]: ak.Array(sum_of_y(array))
Out[  ]: <Array [10, 0, 25, 21] type='4 * int64'>
```

Awkward Array registers its types as known Numba types

expressive for-loop

21 JIT compilation with Numba

```
In [  ]: import awkward as ak; import numpy as np; import numba as nb

In [  ]: array = ak.Array([
    ...:     [{"x": 1.1, "y": [1]}, {"x": 2.2, "y": [1, 2]}, {"x": 3.3, "y": [1, 2, 3]}], # sum'y': 10
    ...:     [],                                                                           # sum'y': 0
    ...:     [{"x": 4.4, "y": [1, 2, 3, 4]}, {"x": 5.5, "y": [1, 2, 3, 4, 5]}],           # sum'y': 25
    ...:     [{"x": 6.6, "y": [1, 2, 3, 4, 5, 6]}],                                       # sum'y': 21
    ...: ])

In [  ]: ak.sum(ak.sum(array.y, axis=-1), axis=-1)
Out[  ]: <Array [10, 0, 25, 21] type='4 * int64'>

In [  ]: @nb.jit
    ...: def sum_of_y(array):
    ...:     out = np.zeros(len(array), dtype=np.int64)
    ...:     for i, list_of_records in enumerate(array):
    ...:         for record in list_of_records:
    ...:             for y in record.y:
    ...:                 out[i] += y
    ...:     return out
    ...:

In [  ]: ak.Array(sum_of_y(array))
Out[  ]: <Array [10, 0, 25, 21] type='4 * int64'>

In [  ]: %timeit ak.sum(ak.sum(array.y, axis=-1), axis=-1)
264 µs ± 7.59 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)

In [  ]: %timeit ak.Array(sum_of_y(array))
18.3 µs ± 352 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

Awkward Array registers its types as known Numba types

expressive for-loop

JIT compilation performance

22 Awkward Array provides all ingredients for HEP

1. **Isolate** interesting tracks (lines)
2. **Combinatorics**: e.g. find parent particles
3. **Reconstruct** physical properties (mass)

12 Jagged Arrays: basics (1/2)

Representation:

User code:

```
energy = ak.Array([
  [50, 100, 30],      # event 1
  [5],                # event 2
  [40, 20],           # event 3
])
```

Memory layout:

```
energy-starts = 0    3  4
energy-stops = 3     4  6

energy-data   = 50, 100, 30, 5, 40, 20
```

Manipulation (“remove first energy from all events”):

User code:

```
energy[:, 1:]
>> ak.Array([
  [100, 30],          # event 1
  [ ],                # event 2
  [20],               # event 3
])
```

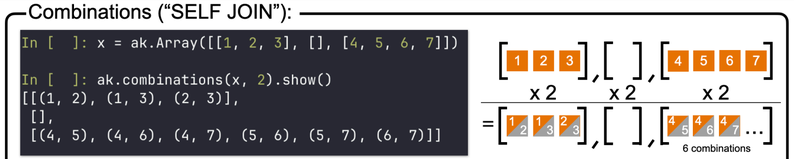
Memory layout:

```
energy-starts = 1    4  5   ←+1
energy-stops = 3     4  6

energy-data   = 50, 100, 30, 5, 40, 20
```

15 Jagged Arrays: combinatorics

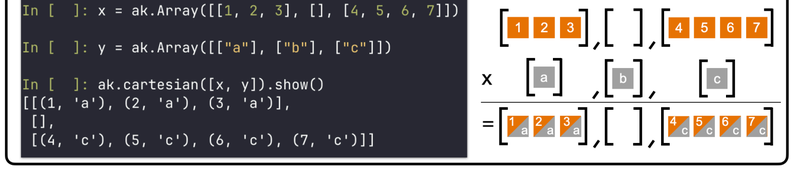
Combinations (“SELF JOIN”):

```
In [  ]: x = ak.Array([[1, 2, 3], [], [4, 5, 6, 7]])

In [  ]: ak.combinations(x, 2).show()
[[(1, 2), (1, 3), (2, 3)],
 [],
 [(4, 5), (4, 6), (4, 7), (5, 6), (5, 7), (6, 7)]]
```



Cartesian product (“CROSS JOIN”):

```
In [  ]: x = ak.Array([[1, 2, 3], [], [4, 5, 6, 7]])

In [  ]: y = ak.Array([["a"], ["b"], ["c"]])

In [  ]: ak.cartesian([x, y]).show()
[[(1, 'a'), (2, 'a'), (3, 'a')],
 [],
 [(4, 'c'), (5, 'c'), (6, 'c'), (7, 'c')]]
```



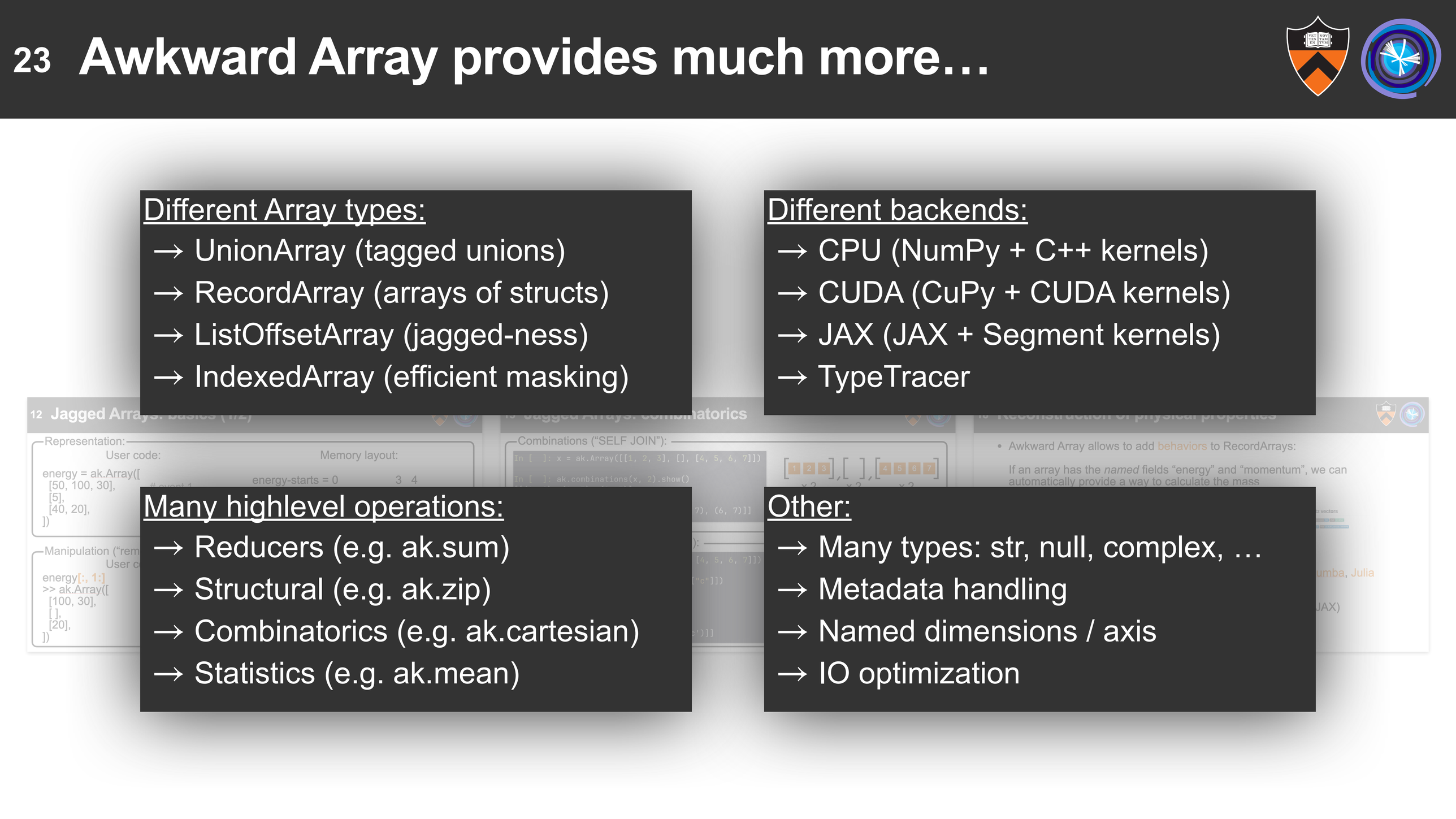
18 Reconstruction of physical properties

- Awkward Array allows to add behaviors to RecordArrays:

  If an array has the *named* fields “energy” and “momentum”, we can automatically provide a way to calculate the mass
- All of these physics quantities are abstracted into behaviors via *vector* (a package to work with Lorentz vectors)

- For more complex reconstructions Awkward Array provides:
  - Interoperability with JIT-compiled for-loop expressions in Numba, Julia (Awkward.jl), and C++ (cppyy)
  - Various conversions to ML libraries (PyTorch, TensorFlow, JAX)

23 Awkward Array provides much more…



Different Array types:

→ UnionArray (tagged unions)
→ RecordArray (arrays of structs)
→ ListOffsetArray (jagged-ness)
→ IndexedArray (efficient masking)

Different backends:

→ CPU (NumPy + C++ kernels)
→ CUDA (CuPy + CUDA kernels)
→ JAX (JAX + Segment kernels)
→ TypeTracer

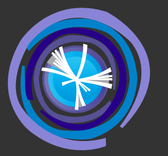
Many highlevel operations:

→ Reducers (e.g. ak.sum)
→ Structural (e.g. ak.zip)
→ Combinatorics (e.g. ak.cartesian)
→ Statistics (e.g. ak.mean)

Other:

→ Many types: str, null, complex, …
→ Metadata handling
→ Named dimensions / axis
→ IO optimization

24 Awkward Array provides much more…



Different Array types:

→ **UnionArray (tagged unions)**

→ **RecordArray (arrays of structs)**

→ **ListOffsetArray (jagged-ness)**

→ **IndexedArray (efficient masking)**

Different backends:

→ CPU (NumPy + C++ kernels)

→ CUDA (CuPy + CUDA kernels)

→ JAX (JAX + Segment kernels)

→ TypeTracer

Many highlevel operations:

→ Reducers (e.g. ak.sum)

→ Structural (e.g. ak.zip)

→ Combinatorics (e.g. ak.cartesian)

→ Statistics (e.g. ak.mean)

Other:

→ Many types: str, null, complex, …

→ Metadata handling

→ Named dimensions / axis

→ IO optimization

Awkward Array: High-level

- High-level array type: **ak.Array**
- Users will only interact with this type
- It looks and feels like a NumPy array
- Supports a (Jupyter) pretty printing
- Type information includes shape & dtype
- Has highlevel metadata for the whole array in **.attrs** (similar to XArray)
- Supports NumPy-like indexing (including fancy indexing with other Awkward Arrays)

```
In [  ]: import awkward as ak

In [  ]: array = ak.Array([
    ...:     [{"x": 1.1, "y": [1]}, {"x": 3.3, "y": [1, 2, 3]}],
    ...:     [],
    ...:     [{"x": 4.4, "y": [1, 2, 3]}, {"x": 5.5, "y": [1, 2, 3]}],
    ...:     [{"x": 6.6, "y": [1, 2, 3, 4, 5, 6]}],
    ...: ])

In [  ]: array.show(all=True)
type: 4 * var * {
    x: float64,
    y: var * int64
}
nbytes: 256 B
backend: cpu
[[{x: 1.1, y: [1]}, {x: 3.3, y: [1, 2, 3]}],
 [],
 [{x: 4.4, y: [1, 2, 3]}, {x: 5.5, y: [1, 2, 3]}],
 [{x: 6.6, y: [1, 2, 3, 4, 5, 6]}]]

In [  ]: array[ak.num(array.y, axis=-1) == 3].show(all=True)
type: 4 * var * {
    x: float64,
    y: var * int64
}
nbytes: 280 B
backend: cpu
[[{x: 3.3, y: [1, 2, 3]}],
 [],
 [{x: 4.4, y: [1, 2, 3]}, {x: 5.5, y: [1, 2, 3]}],
 []]
```

Awkward Array: Layouts

- Layouts are a XML-like internal structure of a highlevel awkward Array
- Layouts implement type-specific logic for jagged arrays (e.g. indexing)
- There are different types of layouts, each with a unique meaning

Adds jaggedness

Adds records

Holds flat data

```
In [  ]: array.layout
Out[  ]:
<ListOffsetArray len='4'>
    <offsets><Index dtype='int64' len='5'>
        [0 2 2 4 5]
    </Index></offsets>
    <content><RecordArray is_tuple='false' len='5'>
        <content index='0' field='x'>
            <NumpyArray dtype='float64' len='5'>[1.1 3.3 4.4 5.5 6.6]</NumpyArray>
        </content>
        <content index='1' field='y'>
            <ListOffsetArray len='5'>
                <offsets><Index dtype='int64' len='6'>
                    [ 0  1  4  7 10 16]
                </Index></offsets>
                <content><NumpyArray dtype='int64' len='16'>
                    [1 1 2 3 1 2 3 1 2 3 1 2 3 4 5 6]
                </NumpyArray></content>
            </ListOffsetArray>
        </content>
    </RecordArray></content>
</ListOffsetArray>
```

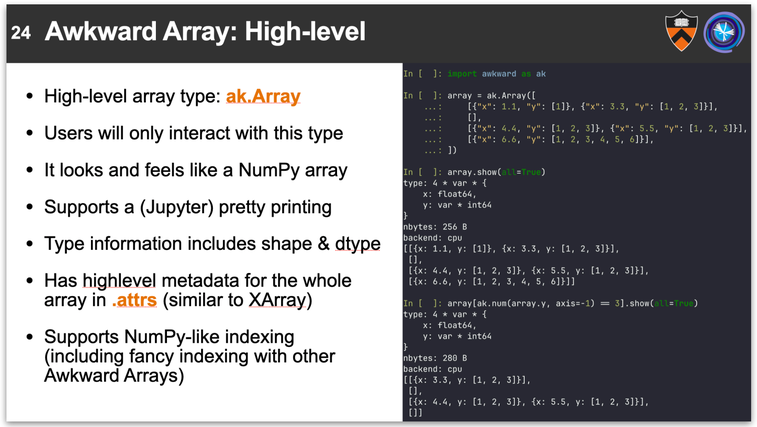
27 Awkward Array: Forms

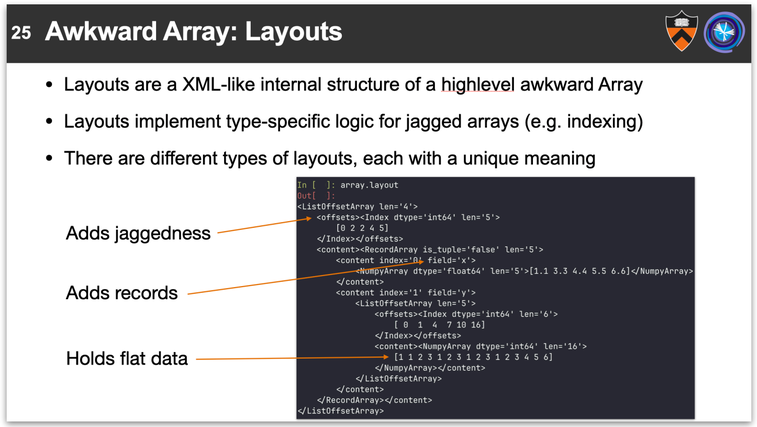
- Layouts can be further decomposed into a “form” and a container of arrays
- Form is a structural “blueprint” of the layout:
  - It’s a JSON serializable tree-like Python structure (nested dicts)
  - It contains per node: layout type, dtype, and metadata
  - Each node has a unique name/identifier “form_key”
- Given a form and a mapping of form keys to arrays we can re-construct an awkward Array

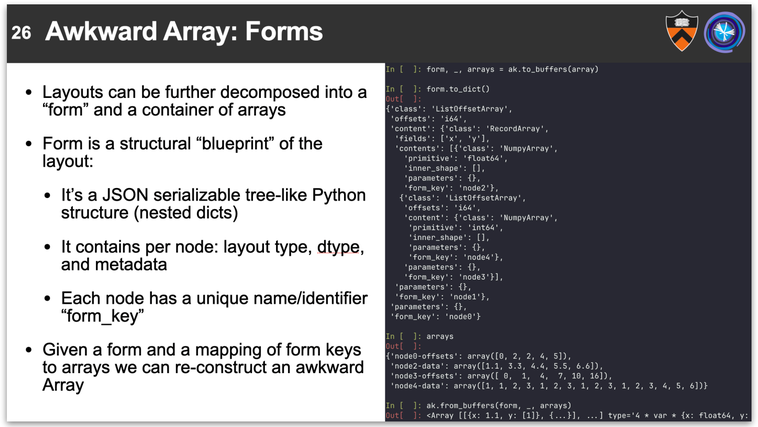
```
In [  ]: form, _, arrays = ak.to_buffers(array)

In [  ]: form.to_dict()
Out[  ]:
{'class': 'ListOffsetArray',
 'offsets': 'i64',
 'content': {'class': 'RecordArray',
  'fields': ['x', 'y'],
  'contents': [{'class': 'NumpyArray',
    'primitive': 'float64',
    'inner_shape': [],
    'parameters': {},
    'form_key': 'node2'},
   {'class': 'ListOffsetArray',
    'offsets': 'i64',
    'content': {'class': 'NumpyArray',
     'primitive': 'int64',
     'inner_shape': [],
     'parameters': {},
     'form_key': 'node4'},
    'parameters': {},
    'form_key': 'node3'}],
  'parameters': {},
  'form_key': 'node1'},
 'parameters': {},
 'form_key': 'node0'}

In [  ]: arrays
Out[  ]:
{'node0-offsets': array([0, 2, 2, 4, 5]),
 'node2-data': array([1.1, 3.3, 4.4, 5.5, 6.6]),
 'node3-offsets': array([ 0,  1,  4,  7, 10, 16]),
 'node4-data': array([1, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 4, 5, 6])}

In [  ]: ak.from_buffers(form, _, arrays)
Out[  ]: <Array [[{x: 1.1, y: [1]}, {...}], ...] type='4 * var * {x: float64, y:
```

28 Awkward Array: Overview of Array Levels

1. **High-Level**: users interact with this
2. **Layouts**: implement jagged logic/ops
3. **Forms**: separate structure from data







29 Awkward Array: Overview of Array Levels

Awkward internals

1. **High-Level**: users interact with this

2. **Layouts**: implement jagged logic/ops

3. **Forms**: separate structure from data

24 Awkward Array: High-level

- High-level array type: ak.Array
- Users will only interact with this type
- It looks and feels like a NumPy array
- Supports a (Jupyter) pretty printing
- Type information includes shape & dtype
- Has highlevel metadata for the whole array in .attrs (similar to XArray)
- Supports NumPy-like indexing (including fancy indexing with other Awkward Arrays)

```
In [ ]: import awkward as ak

In [ ]: array = ak.Array([
   ...:     [{"x": 1.1, "y": [1]}, {"x": 3.3, "y": [1, 2, 3]}],
   ...:     [],
   ...:     [{"x": 4.4, "y": [1, 2, 3]}, {"x": 5.5, "y": [1, 2, 3]}],
   ...:     [{"x": 6.6, "y": [1, 2, 3, 4, 5, 6]}],
   ...: ])

In [ ]: array.show(type=True)
type: 4 * var * {
    x: float64,
    y: var * int64
}
nbytes: 256 B
backend: cpu
[[{x: 1.1, y: [1]}, {x: 3.3, y: [1, 2, 3]}],
 [],
 [{x: 4.4, y: [1, 2, 3]}, {x: 5.5, y: [1, 2, 3]}],
 [{x: 6.6, y: [1, 2, 3, 4, 5, 6]}]]

In [ ]: array[ak.num(array.y, axis=-1) == 3].show(type=True)
type: 4 * var * {
    x: float64,
    y: var * int64
}
nbytes: 280 B
backend: cpu
[[{x: 3.3, y: [1, 2, 3]}],
 [],
 [{x: 4.4, y: [1, 2, 3]}, {x: 5.5, y: [1, 2, 3]}],
 []]
```

25 Awkward Array: Layouts

- Layouts are a XML-like internal structure of a highlevel awkward Array
- Layouts implement type-specific logic for jagged arrays (e.g. indexing)
- There are different types of layouts, each with a unique meaning

Adds jaggedness

Adds records

Holds flat data

```
In [ ]: array.layout
Out[ ]:
<ListOffsetArray len='4'>
    <offsets><Index dtype='int64' len='5'>
        [0 2 2 4 5]
    </Index></offsets>
    <content><RecordArray is_tuple='false' len='5'>
        <content index='0' field='x'>
            <NumpyArray dtype='float64' len='5'>[1.1 3.3 4.4 5.5 6.6]</NumpyArray>
        </content>
        <content index='1' field='y'>
            <ListOffsetArray len='5'>
                <offsets><Index dtype='int64' len='6'>
                    [ 0  1  4  7 10 16]
                </Index></offsets>
                <content><NumpyArray dtype='int64' len='16'>
                    [1 1 2 3 1 2 3 1 2 3 1 2 3 4 5 6]
                </NumpyArray></content>
            </ListOffsetArray>
        </content>
    </RecordArray></content>
</ListOffsetArray>
```

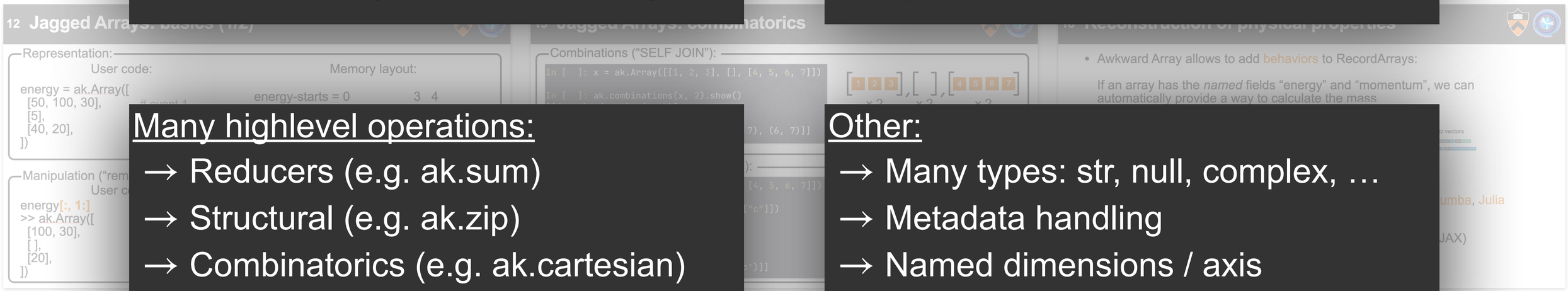
26 Awkward Array: Forms

- Layouts can be further decomposed into a “form” and a container of arrays
- Form is a structural “blueprint” of the layout:
  - It’s a JSON serializable tree-like Python structure (nested dicts)
  - It contains per node: layout type, dtype, and metadata
  - Each node has a unique name/identifier “form_key”
- Given a form and a mapping of form keys to arrays we can re-construct an awkward Array

```
In [ ]: form, _, arrays = ak.to_buffers(array)

In [ ]: form.to_dict()
Out[ ]:
{'class': 'ListOffsetArray',
 'offsets': 'i64',
 'content': {'class': 'RecordArray',
  'fields': ['x', 'y'],
  'contents': [{'class': 'NumpyArray',
    'primitive': 'float64',
    'inner_shape': [],
    'parameters': {},
    'form_key': 'node2'},
   {'class': 'ListOffsetArray',
    'offsets': 'i64',
    'content': {'class': 'NumpyArray',
     'primitive': 'int64',
     'inner_shape': [],
     'parameters': {},
     'form_key': 'node4'},
    'parameters': {},
    'form_key': 'node3'}],
  'parameters': {},
  'form_key': 'node1'},
 'parameters': {},
 'form_key': 'node0'}

In [ ]: arrays
Out[ ]:
{'node0-offsets': array([0, 2, 2, 4, 5]),
 'node2-data': array([1.1, 3.3, 4.4, 5.5, 6.6]),
 'node3-offsets': array([ 0,  1,  4,  7, 10, 16]),
 'node4-data': array([1, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 4, 5, 6])}

In [ ]: ak.from_buffers(form, _, arrays)
Out[ ]: <Array [[{x: 1.1, y: [1]}, {...}], ...] type='4 * var * {x: float64, y:
```

(accessible, but usually not needed)

30 Awkward Array provides much more…



Different Array types:

→ UnionArray (tagged unions)

→ RecordArray (arrays of structs)

→ ListOffsetArray (jagged-ness)

→ IndexedArray (efficient masking)

Different backends:

→ **CPU (NumPy + C++ kernels)**

→ **CUDA (CuPy + CUDA kernels)**

→ **JAX (JAX + Segment kernels)**

→ **TypeTracer**

Many highlevel operations:

→ Reducers (e.g. ak.sum)

→ Structural (e.g. ak.zip)

→ Combinatorics (e.g. ak.cartesian)

→ Statistics (e.g. ak.mean)

Other:

→ Many types: str, null, complex, …

→ Metadata handling

→ Named dimensions / axis

→ IO optimization

What's a backend for Awkward Array?

Manages memory of arrays

Implements kernels

One line to choose a backend:

```
ak.Array([[1, 2], [3], [4, 5, 6]], backend=...)
# or
ak.to_backend(array, backend=...)
```

Backend ”cpu”

Manages memory of arrays
NumPy

Implements kernels
NumPy + awkward-cpp

- Runs fully on **CPU**
- This is the **default** backend for Awkward Array
- **awkward-cpp** implements additional kernels that do not exist in NumPy explicitly for jagged arrays

```
ak.Array([[1, 2], [3], [4, 5, 6]], backend="cpu")
```

Backend “cuda”

Manages memory of arrays
CuPy

Implements kernels
CuPy + CUDA

- Runs fully on **GPU**
- Awkward Array provides **jagged CUDA kernels** that do not exist in CuPy explicitly needed for jagged arrays
- **numba.cuda** can be used for custom kernels

```
ak.Array([[1, 2], [3], [4, 5, 6]], backend="cuda")
```

Backend “jax”

Manages memory of arrays
JAX

Implements kernels
JAX

- Can run on **CPU or GPU** (depends on the jax.device)
- We’re using **JAX’s segment kernels** to for jagged kernels
- Does support **auto-differentiation**, but no JIT-compilation

```
ak.Array([[1, 2], [3], [4, 5, 6]], backend="jax")
```

Backend “typetracer”

Manages memory of arrays
TypeTracerArray

Implements kernels
TypeTracer

- TypeTracerArrays are Python objects that **look and behave like Arrays**, but **do not contain any numerical values**
- “TypeTracer” module implements a subset of Array-API-like methods to work with TypeTracerArrays
- more on them later…

```
ak.Array([[1, 2], [3], [4, 5, 6]], backend="typetracer")
```

36 Awkward Array provides much more…

Different Array types:

→ UnionArray (tagged unions)

→ RecordArray (arrays of structs)

→ ListOffsetArray (jagged-ness)

→ IndexedArray (efficient masking)

Different backends:

→ CPU (NumPy + C++ kernels)

→ CUDA (CuPy + CUDA kernels)

→ JAX (JAX + Segment kernels)

→ TypeTracer

Many highlevel operations:

→ Reducers (e.g. ak.sum)

→ Structural (e.g. ak.zip)

→ Combinatorics (e.g. ak.cartesian)

→ Statistics (e.g. ak.mean)

Other:

→ Many types: str, null, complex, …

→ Metadata handling

→ **Named dimensions / axis**

→ IO optimization

Named axis: benefits

More readable & understandable code (wished by many supervisors in HEP)

Less errors: wrong axis, wrong broadcastings, …

New safety opportunities: binary operations that involve 2 axes with different names can throw an error (i.e. you shouldn't add "events" and "particles")

Nice writeup by HarvardNLP team why named axis are important:

Tensor Considered Harmful

Alexander Rush - @harvardnlp

Open in Colab

TL;DR: Despite its ubiquity in deep learning, Tensor is broken. It forces bad habits such as exposing private dimensions, broadcasting based on absolute position, and keeping type information in documentation. This post presents a proof-of-concept of an alternative approach, ***named tensors****, with named dimensions. This change eliminates the need for indexing, dim arguments, einsum- style unpacking, and documentation-based coding. The prototype* ***PyTorch library*** *accompanying this blog post is available as namedtensor.*

https://nlp.seas.harvard.edu/NamedTensor.html

Named axis: basics

Attaching named axis:

```
In [  ]: x = ak.Array([[1, 2], [3, 4]], named_axis=("events", "particles"))

In [  ]: x = ak.Array([[1, 2], [3, 4]], named_axis={"events": 0, "particles": 1})

In [  ]: x = ak.with_named_axis(ak.Array([[1, 2], [3, 4]]), named_axis=("events", "particles"))

In [  ]: x = ak.with_named_axis(ak.Array([[1, 2], [3, 4]]), named_axis={"events": 0, "particles": 1})
```

Use names for reductions:

```
In [  ]: ak.sum(x, axis="particles")
Out[  ]: <Array [3, 7] events:0 type='2 * int64'>

In [  ]: np.sum(x, axis="particles")
Out[  ]: <Array [3, 7] events:0 type='2 * int64'>

In [  ]: ak.sum(x, axis="events")
Out[  ]: <Array [4, 6] particles:0 type='2 * int64'>
```

It’s much more than just a name to dimension mapping:

named axis dynamically adjust with all awkward ops

Named axis: indexing (1/3)

Indexing adjusts named axes as well:

```
In [   ]: x
Out[   ]: <Array [[1, 2], [3, 4]] events:0,particles:1 type='2 * var * int64'>

In [   ]: x[:, 0]
Out[   ]: <Array [1, 3] events:0 type='2 * int64'>

In [   ]: x[:, 1]
Out[   ]: <Array [2, 4] events:0 type='2 * int64'>

In [   ]: x[0, :]
Out[   ]: <Array [1, 2] particles:0 type='2 * int64'>

In [   ]: x[1, :]
Out[   ]: <Array [3, 4] particles:0 type='2 * int64'>

In [   ]: x[None, ...]
Out[   ]: <Array [[[1, 2], [3, 4]]] events:1,particles:2 type='1 * 2 * var * int64'>
```

Named axis: indexing (2/3)

New indexing syntax: dictionary-style indexing with named axes

```
In [  ]: x
Out[  ]: <Array [[1, 2], [3, 4]] events:0,particles:1 type='2 * var * int64'>

In [  ]: x[{"events": 0}]
Out[  ]: <Array [1, 2] particles:0 type='2 * int64'>

In [  ]: x[{"events": 1}]
Out[  ]: <Array [3, 4] particles:0 type='2 * int64'>

In [  ]: x[{"particles": 0}]
Out[  ]: <Array [1, 3] events:0 type='2 * int64'>

In [  ]: x[{"particles": 1}]
Out[  ]: <Array [2, 4] events:0 type='2 * int64'>
```

Named axis: indexing (3/3)

New indexing syntax (advanced): mixed positional & named axis indexing

```
In [  ]: x
Out[  ]: <Array [[1, 2], [3, 4]] events:0,particles:1 type='2 * var * int64'>

In [  ]: x[{0: 0}]
Out[  ]: <Array [1, 2] particles:0 type='2 * int64'>

In [  ]: x[{0: 1}]
Out[  ]: <Array [3, 4] particles:0 type='2 * int64'>

In [  ]: x[{"particles": np.s_[0:1]}]
Out[  ]: <Array [[1], [3]] events:0,particles:1 type='2 * var * int64'>

In [  ]: x[{0: 0, "particles": np.s_[0:1]}]
Out[  ]: <Array [1] particles:0 type='1 * int64'>
```

Named axis: safety guards

Safety guard: conflicting axes names will error

```
In [  ]: x = ak.Array([[1, 2], [3, 4]], named_axis=("events", "electrons"))

In [  ]: y = ak.Array([[1, 2], [3, 4]], named_axis=("events", "muons"))

In [  ]: x + y
---------------------------------------------------------------------------
ValueError: The named axes are incompatible. Got: electrons and muons for positional axis 1

This error occurred while calling

    numpy.add.__call__(
        <Array [[1, 2], [3, ...]] events:0,electrons:1 type='2 * var * int64'>
        <Array [[1, 2], [3, 4]] events:0,muons:1 type='2 * var * int64'>
    )
```

43 Summary: Awkward Array & Named Axis

Awkward Array: general-purpose array library for jagged data

12 Jagged Arrays: basics (1/2)

Representation:

User code:

```
energy = ak.Array([
  [50, 100, 30],      # event 1
  [5],                # event 2
  [40, 20],           # event 3
])
```

Memory layout:

```
energy-starts = 0        3  4
energy-stops = 3         4  6

energy-data   = 50, 100, 30, 5, 40, 20
```

Manipulation (“remove first energy from all events”):

User code:

```
energy[:, 1:]
>> ak.Array([
  [100, 30],          # event 1
  [ ],                # event 2
  [20],               # event 3
])
```

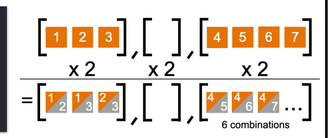
Memory layout:

```
energy-starts = 1        4  5   ←+1
energy-stops = 3         4  6

energy-data   = 50, 100, 30, 5, 40, 20
```

15 Jagged Arrays: combinatorics

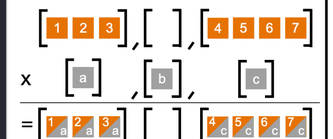
Combinations (“SELF JOIN”):

```
In [  ]: x = ak.Array([[1, 2, 3], [], [4, 5, 6, 7]])

In [  ]: ak.combinations(x, 2).show()
[[(1, 2), (1, 3), (2, 3)],
 [],
 [(4, 5), (4, 6), (4, 7), (5, 6), (5, 7), (6, 7)]]
```



Cartesian product (“CROSS JOIN”):

```
In [  ]: x = ak.Array([[1, 2, 3], [], [4, 5, 6, 7]])

In [  ]: y = ak.Array([["a"], ["b"], ["c"]])

In [  ]: ak.cartesian([x, y]).show()
[[(1, 'a'), (2, 'a'), (3, 'a')],
 [],
 [(4, 'c'), (5, 'c'), (6, 'c'), (7, 'c')]]
```



18 Reconstruction of physical properties

- Awkward Array allows to add behaviors to RecordArrays:

  If an array has the *named* fields “energy” and “momentum”, we can automatically provide a way to calculate the mass
- All of these physics quantities are abstracted into behaviors via *vector* (a package to work with Lorentz vectors)

- For more complex reconstructions Awkward Array provides:
  - Interoperability with JIT-compiled for-loop expressions in Numba, Julia (Awkward.jl), and C++ (cppyy)
  - Various conversions to ML libraries (PyTorch, TensorFlow, JAX)

New feature: *named axis*

Named axis ops

```
ak.sum(
    array,
    axis=“particles”,
)
```

Named indexing

```
array[{“particles”: 0}]
```

Safety guards

```
x + y
>> ValueError:
>> “Incompatible axes”
```

High energy physics: computational view

- Large amounts of data
  - 40 million events (“collisions”) per second
  - Each event is ~1MB of data → 40TB/s data rate

Distributed computing

- Every event is independent
- Every event produces a *variable* number of particles
- Detectors measure a collection of information *per particle*: momentum, energy, position, …
- Combining decay products into parent particles

HEP analysis computational challenges

A lot of data:

- total $\mathcal{O}(10 - 100)$ TB (*after* central selections & reprocessing)
- split in many files each of ~2GB (compressed) size

→ need to scale to clusters to process this data in parallel

Each file contains $\mathcal{O}(1000)$ columns (i.e. physical quantities: energy, mass, …)

- but often only a subset ($\mathcal{O}(100)$) are of interest
- and usually memory is very limited per worker

→ need a mechanism for loading only the required columns (IO-optimization)

dask *and* “Array tracing”

HEP analysis computational challenges

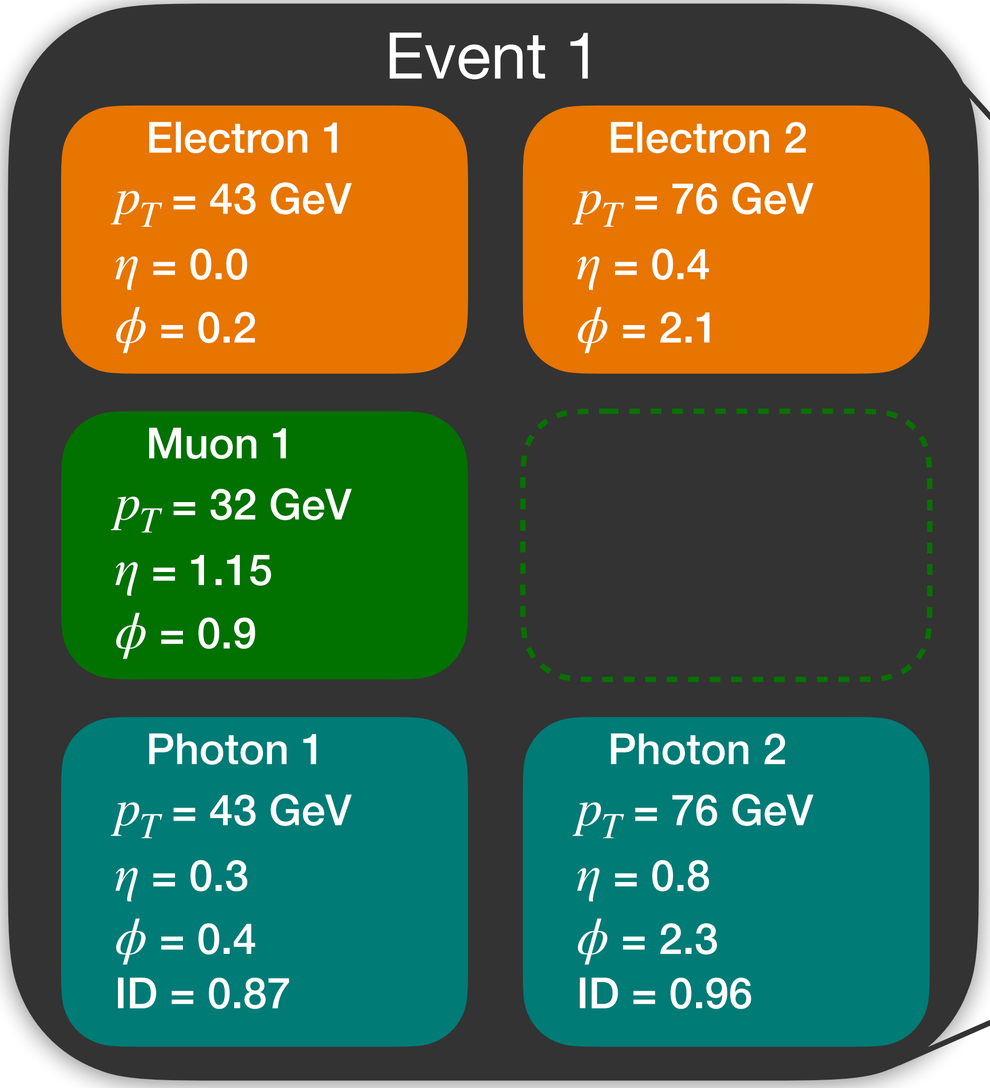
A lot of data:

- total $\mathcal{O}(10 - 100)$ TB (*after* central selections & reprocessing)
- split in many files each of ~2GB (compressed) size

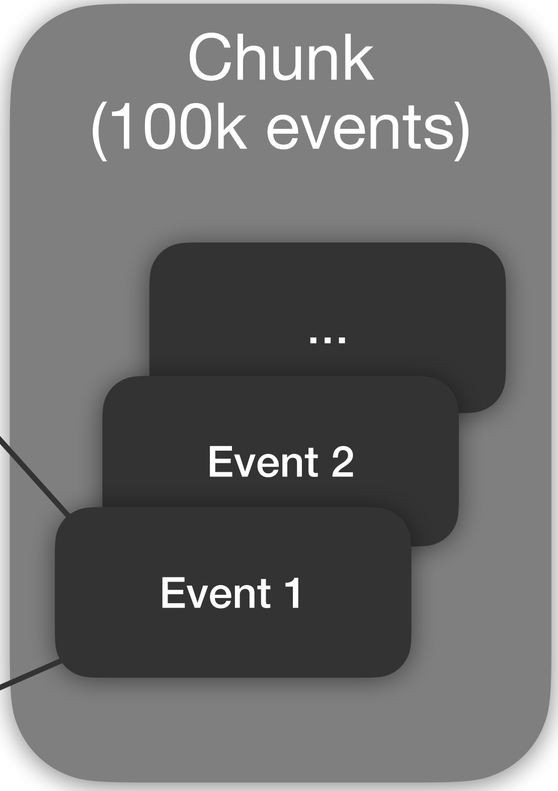
→ need to scale to clusters to process this data in parallel

47 Scaling to clusters with Dask

On a single worker

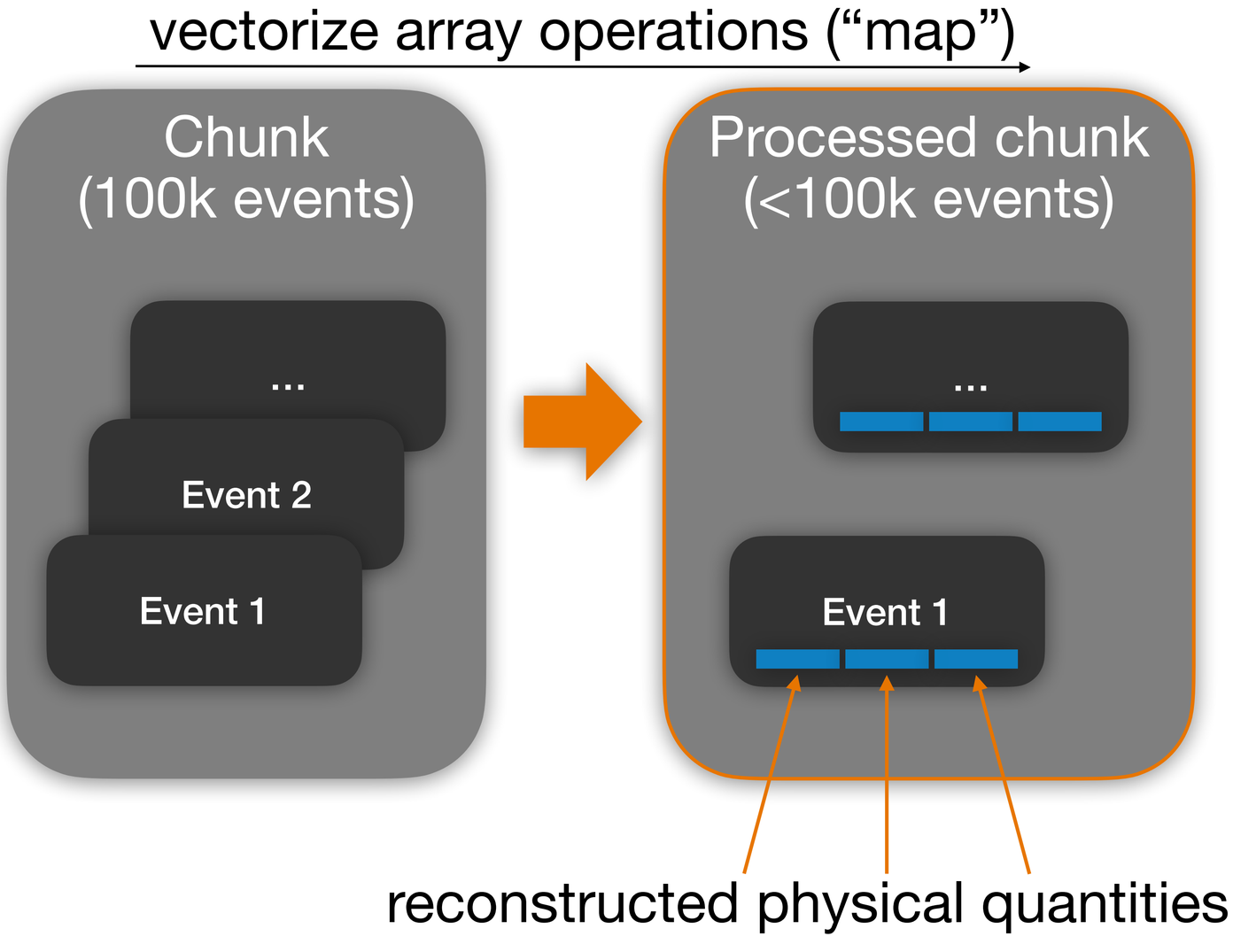


"unit" of data that's processed simultaneously on a single worker



Scaling to clusters with Dask

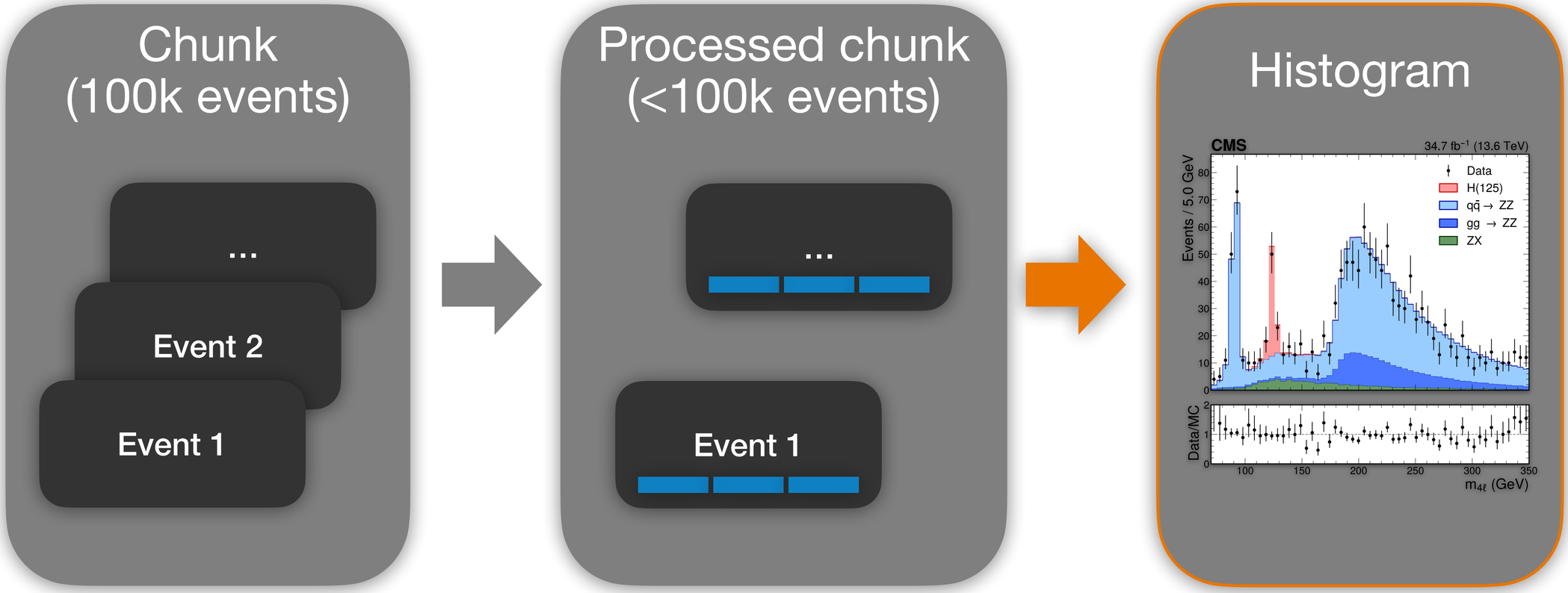
On a single worker



49 Scaling to clusters with Dask

On a single worker

Histogramming (“reduction”)



50 Scaling to clusters with Dask

On a single worker

Map **Reduce**

Chunk
(100k events)

...

Event 2

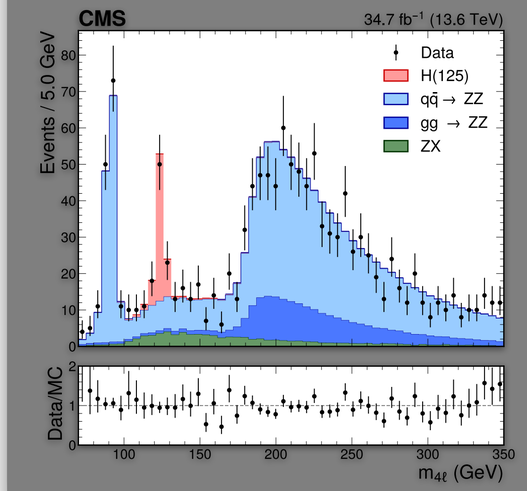
Event 1

Processed chunk
(<100k events)

...

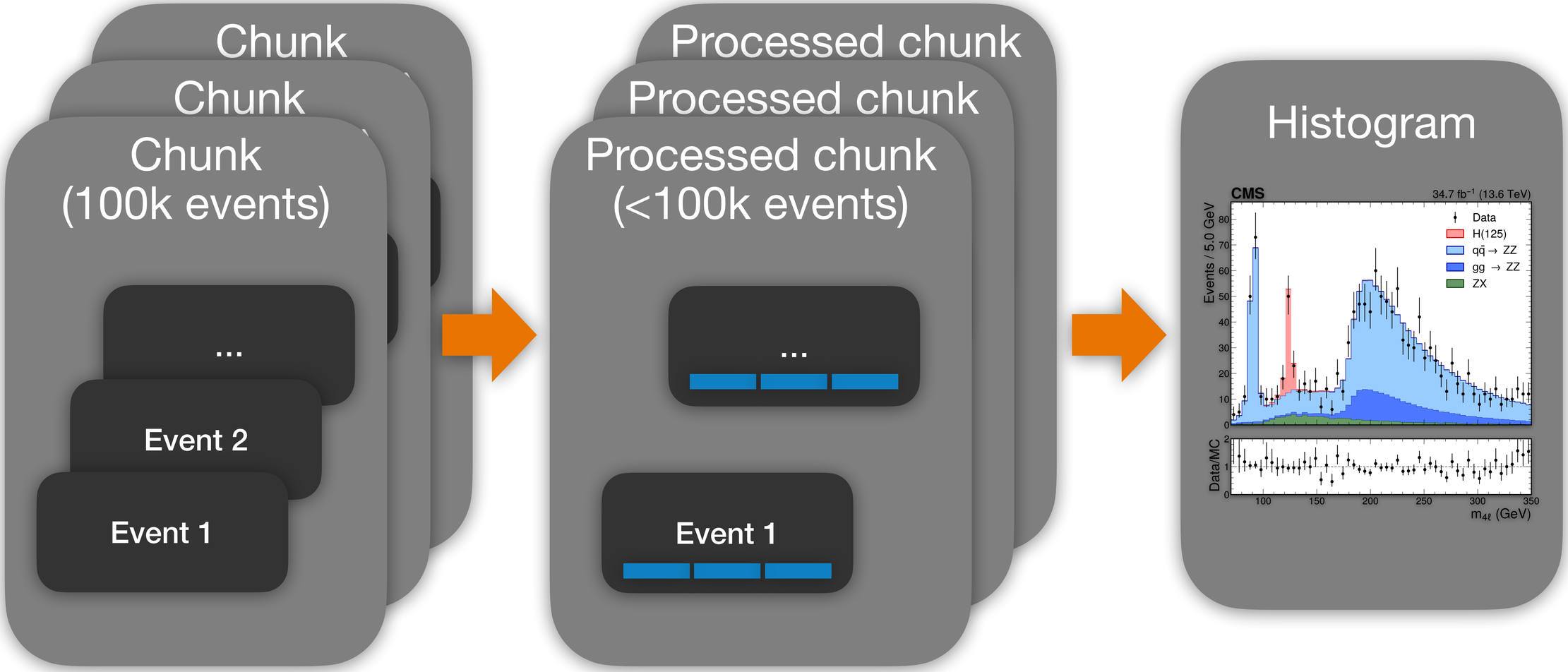
Event 1

Histogram



Scaling to clusters with Dask

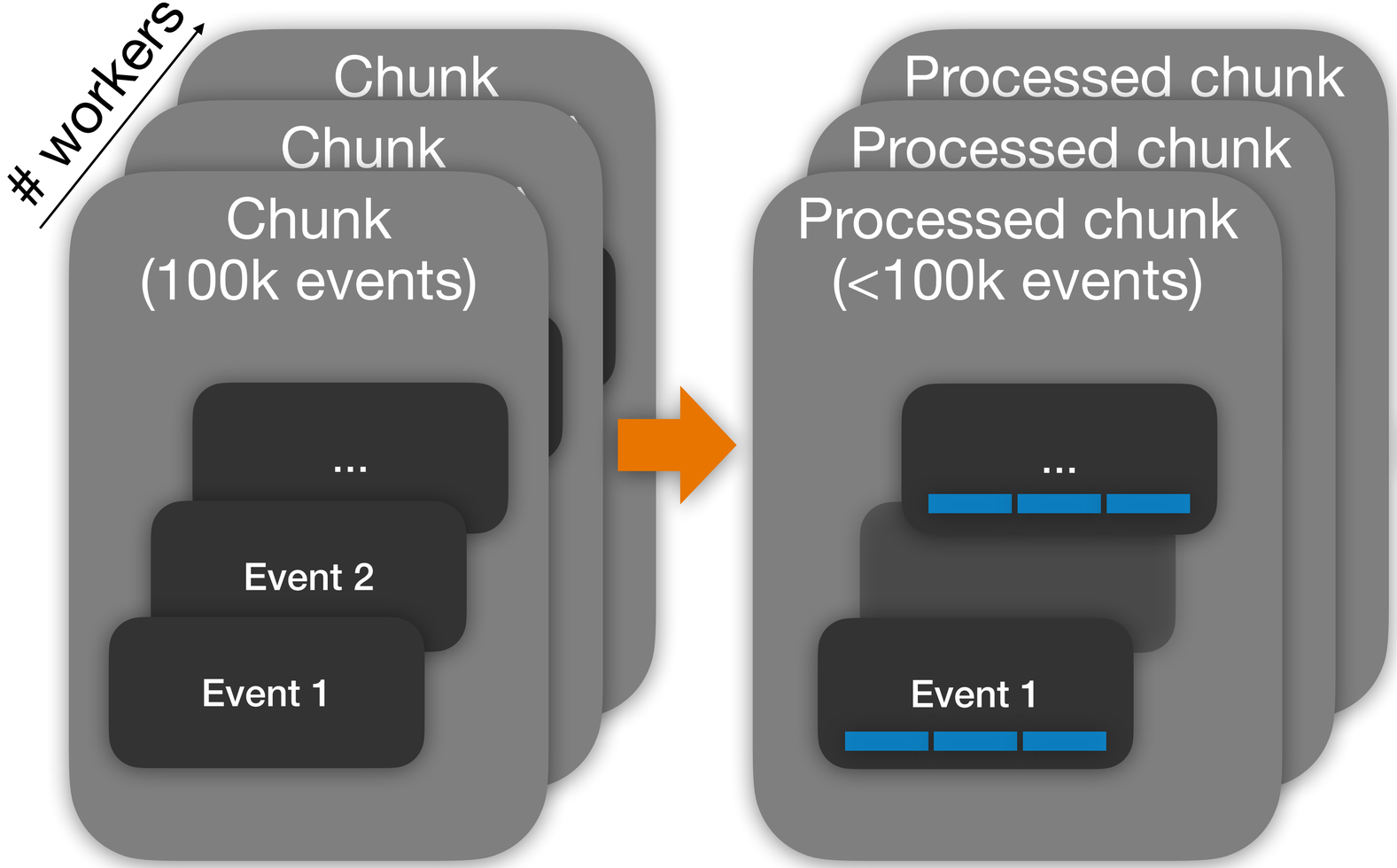
On a cluster (multiple worker)

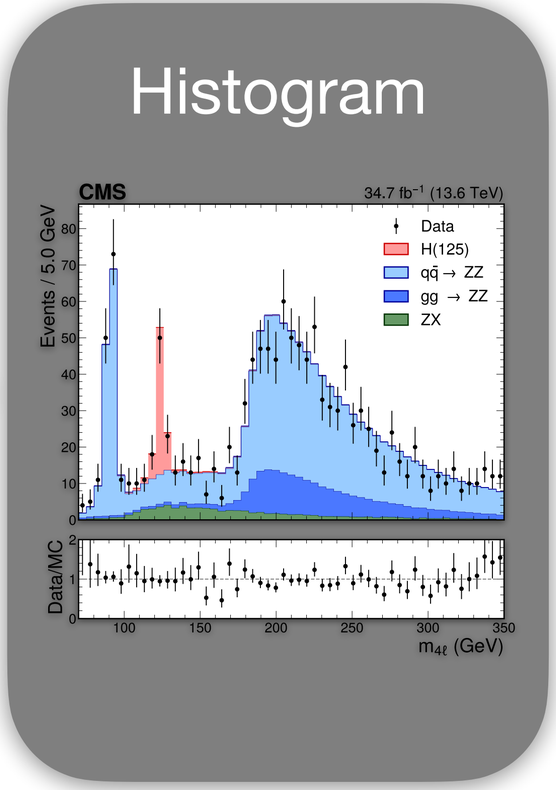


52 Scaling to clusters with Dask

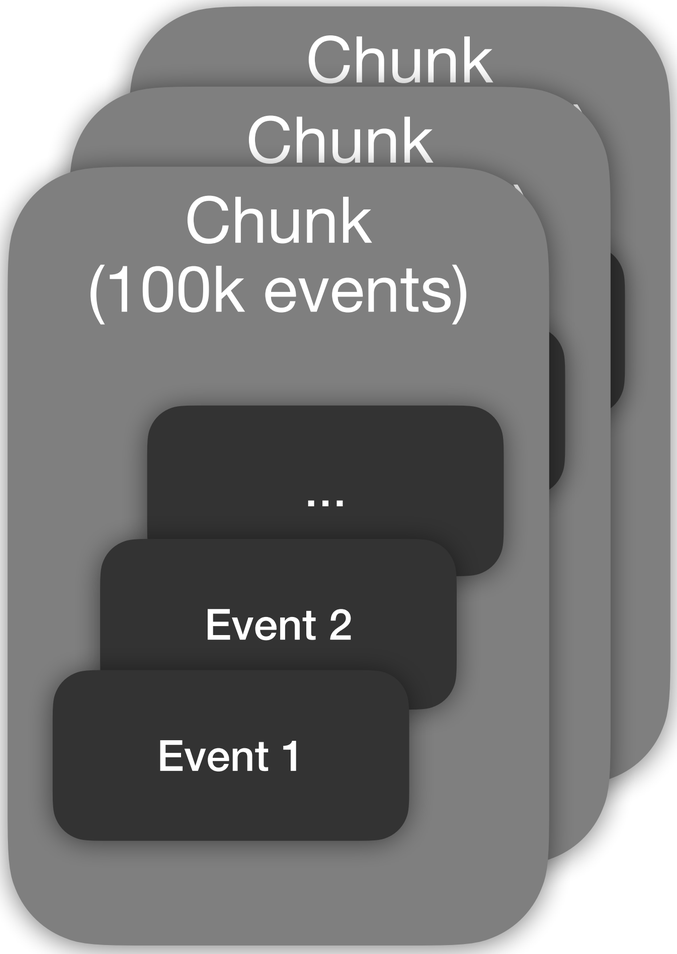
On a cluster (multiple worker)

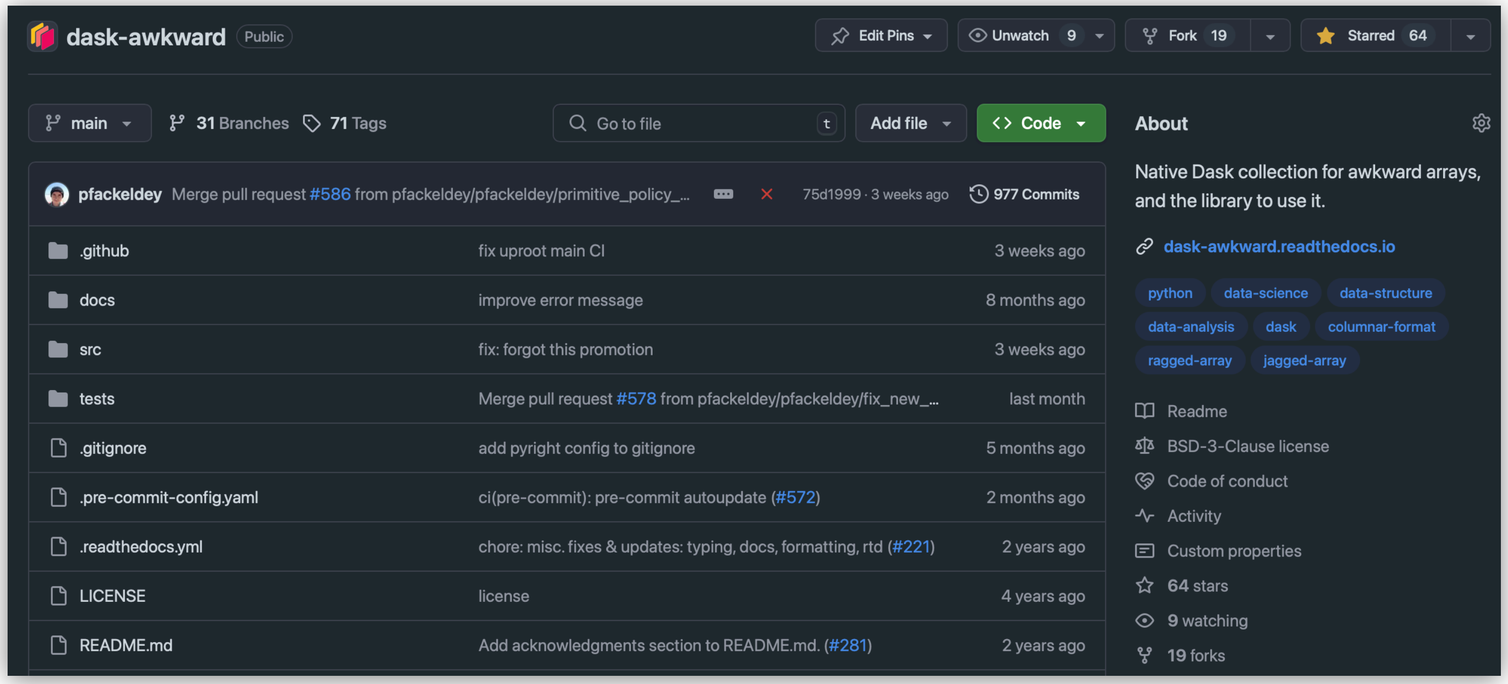
reduction on the client
(no memory constraint)





53 dask-awkward



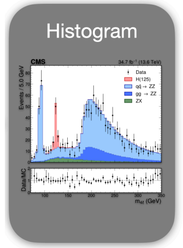


Connect Dask & Awkward Array

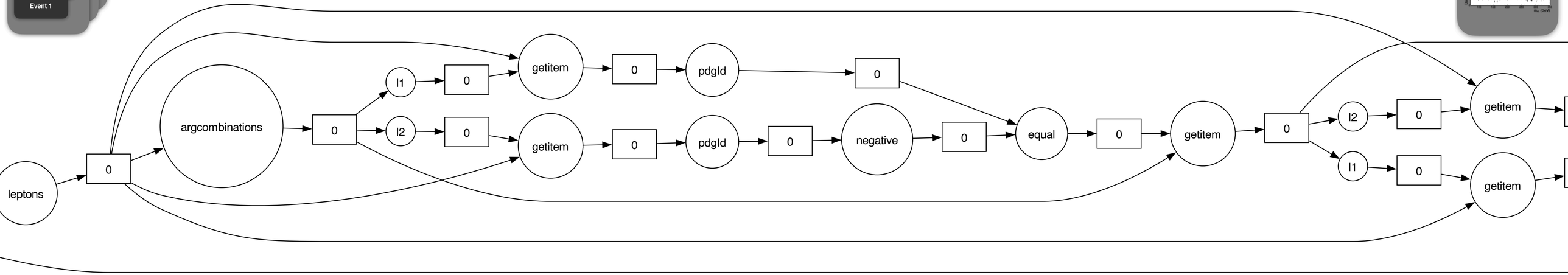
→ build *chunked* Awkward collections

→ build complex task graphs (“map” & ”reduce”)

54 dask-awkward: task graphs

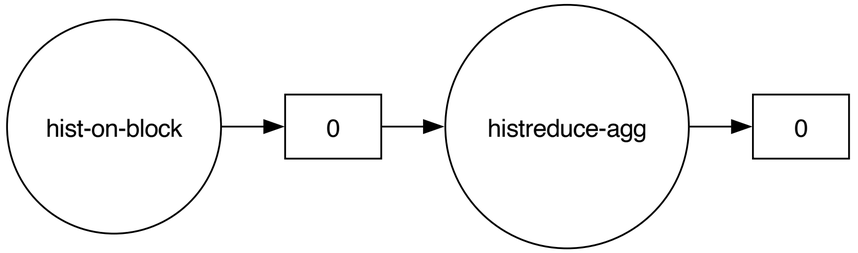


Highly granular HEP task graphs $\mathcal{O}(100)$ layers



Optimized into two “map” & “reduce” layer

This is the graph that’s scheduled on the cluster



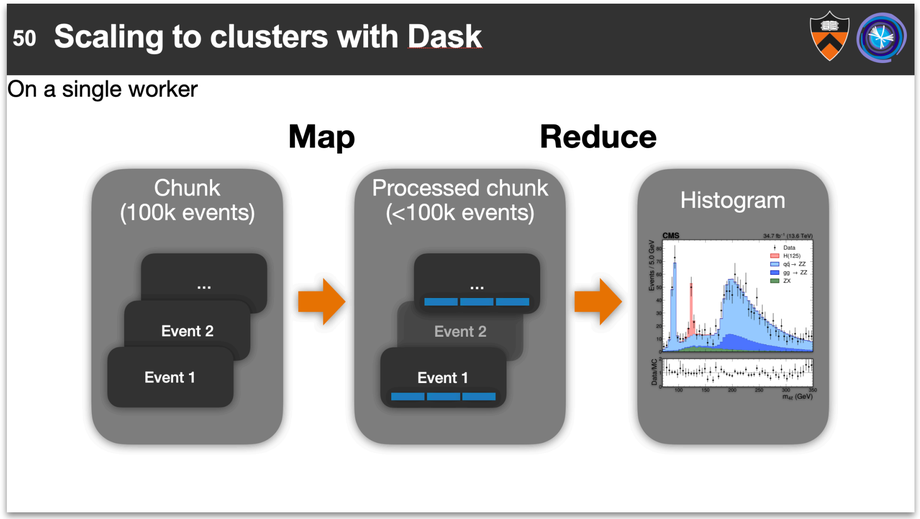
55 Overview: Scaling to clusters

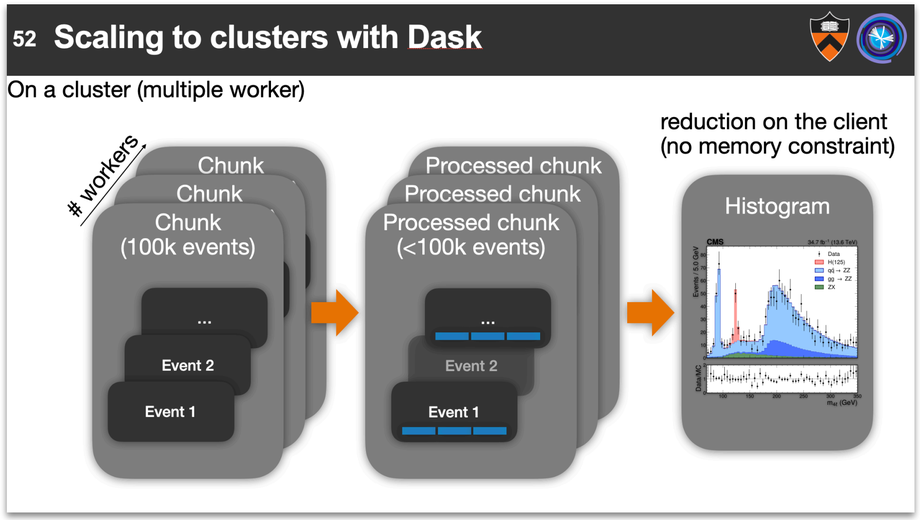
From a single worker viewpoint:

- Working on a “chunk” of events
- Implicit vectorization (accelerated by SIMD) with array paradigms
- Data locality: RAM, CPU caches

From a cluster viewpoint:

- Working on multiple chunks of events
- Parallelization across multiple workers
- Data locality: data is streamed via network into each worker for processing





HEP analysis computational challenges

A lot of data:

- total $\mathcal{O}(10 - 100)$ TB (*after* central selections & reprocessing)
- split in many files each of ~2GB (compressed) size

Each file contains $\mathcal{O}(1000)$ columns (i.e. physical quantities: energy, mass, …)

- but often only a subset ($\mathcal{O}(100)$) are of interest
- and usually memory is very limited per worker

→ need a mechanism for loading only the required columns (IO-optimization)

dask *and* “Array tracing”

57 Array Tracing

What does this function do?

```
def fun(x):
  return x + 1
```

Array Tracing

Ref: Intro to JAX: Accelerating Machine Learning research (adapted)

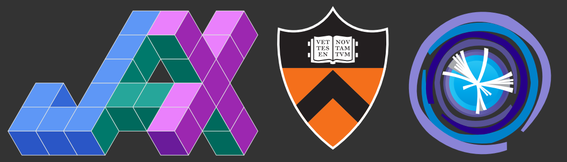
What does this function do?

```python
def fun(x):
  return x + 1


class IncreaseBrightness:
  def __add__(self, x)
    subprocess.call(["ssh", "my@home", ...])


fun(IncreaseBrightness())
```

Python can literally do *anything*

We need to find out **what happens inside *fun*** (without loading any data)

Array Tracing

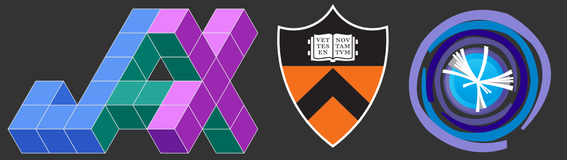


Ref: Intro to JAX: Accelerating Machine Learning research (adapted)

How do tracers work?

```
def fun(x):
  return x + 1


class Tracer:
  recording = []

  def __init__(self, shape, dtype):
    self.shape = shape
    self.dtype = dtype

  def __add__(self, other)
    self.recording.append("add", self, other)
    return Tracer(self.shape, self.dtype)


fun(Tracer(shape=(10,), dtype="float"))
```

Array Tracing



Ref: Intro to JAX: Accelerating Machine Learning research (adapted)

How do tracers work?

```
def fun(x):
  return x + 1


class Tracer:
  recording = []

  def __init__(self, shape, dtype):
    self.shape = shape
    self.dtype = dtype

  def __add__(self, other)
    self.recording.append(        self, other)
    return Tracer(self.shape, self.dtype)


fun(Tracer(shape=(10,), dtype="float"))
```

Awkward Array does record which data is “touched”, *not* the computation itself

TypeTracer (IO-optimization)

- Awkward Array implements its own tracer backend: “TypeTracer”
- TypeTracer records **only** what data has been “touched” by a computation (it does **not record the computations** itself, unlike e.g. JAX’s tracing system)
- Awkward’s tracing procedure:

  1. Delay the *actual* computation (e.g. when using dask)
  2. Run the computation once with a TypeTracer instead of a chunk of data
  3. Load only necessary columns from disk based on tracing information
  4. Run the actual computation only with the loaded columns

  → automatically part of the task graph optimization with dask-awkward

Array Tracing Drawbacks

Tracers do not know about their data (only about their shape and dtype)

→ we can't do data-dependent operations

Tracers can only follow one path of a if-else statement

→ need to trace both branches (e.g. using "where" statements)

We need to trace exactly the computation that would happen with actual data

→ traced functions should be pure (can't depend on some global state)

Can be tricky to debug if tracing fails & physicists are not necessarily familiar with the concept of tracing

→ requires support and teaching about a new concept for analysts

VirtualArrays (IO-optimization)

- The idea: lazy loading of individual RecordArray inside of an Awkward Array
- Whenever a computation needs data, Awkward Array loads it
- Implemented at the lowest internal level (form + array container)
- Each array in the container can be replaced by a callable that delivers the array whenever Awkward needs it
  (~somewhat like a generic mmap for arrays)
- VirtualArray are much more robust than tracing

  …however, we're loosing the knowledge of what we need *before* the actual computation happens

  → we're loosing the ability for most optimal data access
  (maybe important if data is streamed over network)

Currently in R&D

64 VirtualArray: Example

Currently in R&D

```
In [ ]:      arrays = {
    ...:  'node0-offsets':        np.array([0, 2, 2, 4, 5]),
    ...:  'node2-data':          np.array([1.1, 3.3, 4.4, 5.5, 6.6]),
    ...:  'node3-offsets':        np.array([ 0,  1,  4,  7, 10, 16]),
    ...:  'node4-data':          np.array([1, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 4, 5, 6])
    ...: }

In [ ]:       array = ak.from_buffers(form, _, lazy_arrays)
```

65 VirtualArray: Example

Currently in R&D

```
In [  ]: lazy_arrays = {
    ...:  'node0-offsets': lambda: np.array([0, 2, 2, 4, 5]),
    ...:  'node2-data': lambda: np.array([1.1, 3.3, 4.4, 5.5, 6.6]),
    ...:  'node3-offsets': lambda: np.array([ 0,  1,  4,  7, 10, 16]),
    ...:  'node4-data': lambda: np.array([1, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 4, 5, 6])
    ...: }

In [  ]: virtual_array = ak.from_buffers(form, _, lazy_arrays)
```

66 VirtualArray: Example

Currently in R&D

```
In [  ]: lazy_arrays = {
    ...:  'node0-offsets': lambda: np.array([0, 2, 2, 4, 5]),
    ...:  'node2-data': lambda: np.array([1.1, 3.3, 4.4, 5.5, 6.6]),
    ...:  'node3-offsets': lambda: np.array([ 0,  1,  4,  7, 10, 16]),
    ...:  'node4-data': lambda: np.array([1, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 4, 5, 6])
    ...: }

In [  ]: virtual_array = ak.from_buffers(form, _, lazy_arrays)

In [  ]: virtual_array
Out[  ]: <Array [??, ??, ??, ??] type='4 * var * {x: float64, y: var * int64}'>

In [  ]: virtual_array.y
Out[  ]: <Array [??, ??, ??, ??] type='4 * var * var * int64'>

In [  ]: ak.sum(virtual_array.y, axis=-1)
Out[  ]: <Array [[1, 6], [], [6, 6], [21]] type='4 * var * int64'>

In [  ]: virtual_array.y
Out[  ]: <Array [[[1], [1, 2, 3]], ..., [[1, 2, ..., 6]]] type='4 * var * var * int64'>

In [  ]: virtual_array
Out[  ]: <Array [[{x: ??, y: [1]}, {...}], ..., [{...}]] type='4 * var * {x: float64...'>
```

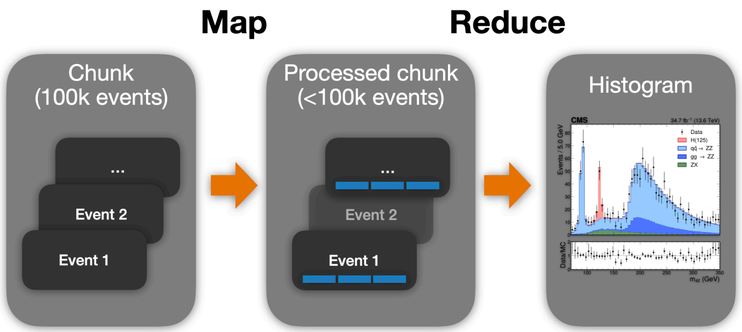
67 Overview: distributed Awkward Array

Parallelization

On a single worker: Awkward Array

50 Scaling to clusters with Dask

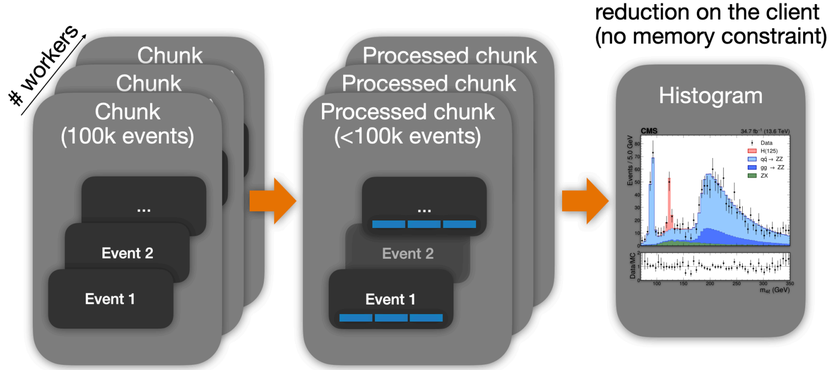
On a single worker



On a cluster: dask (dask-awkward)

52 Scaling to clusters with Dask

On a cluster (multiple worker)



IO optimization

Ahead-of-time: Array Tracing

61 TypeTracer (IO-optimization)

- Awkward Array implements its own tracer backend: “TypeTracer”
- TypeTracer records **only** what data has been “touched” by a computation (it does **not record the computations** itself, unlike e.g. JAX’s tracing system)
- Awkward’s tracing procedure:
  1. Delay the *actual* computation (e.g. when using dask)
  2. Run the computation once with a TypeTracer instead of a chunk of data
  3. Load only necessary columns from disk based on tracing information
  4. Run the actual computation only with the loaded columns

→ automatically part of the task graph optimization with dask-awkward

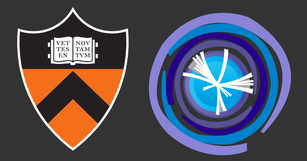
On-demand: Virtual Arrays

66 VirtualArray: Example

Currently in R&D

```
In [ ]: lazy_arrays = {
   ...:  'node0-offsets': lambda: np.array([0, 2, 2, 4, 5]),
   ...:  'node2-data': lambda: np.array([1.1, 3.3, 4.4, 5.5, 6.6]),
   ...:  'node3-offsets': lambda: np.array([ 0,  1,  4,  7, 10, 16]),
   ...:  'node4-data': lambda: np.array([1, 1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 4, 5, 6])
   ...: }

In [ ]: virtual_array = ak.from_buffers(form, _, lazy_arrays)

In [ ]: virtual_array
Out[ ]: <Array [??, ??, ??, ??] type='4 * var * {x: float64, y: var * int64}'>

In [ ]: virtual_array.y
Out[ ]: <Array [??, ??, ??, ??] type='4 * var * var * int64'>

In [ ]: ak.sum(virtual_array.y, axis=-1)
Out[ ]: <Array [[1, 6], [], [6, 6], [21]] type='4 * var * int64'>

In [ ]: virtual_array.y
Out[ ]: <Array [[[1], [1, 2, 3]], ..., [[1, 2, ..., 6]]] type='4 * var * var * int64'>

In [ ]: virtual_array
Out[ ]: <Array [[{x: ??, y: [1]}, {...}], ..., [{...}]] type='4 * var * {x: float64...'>
```

Summary



Awkward Array:

- Generic array library for **ragged arrays**
- Focusses on **working** with ragged arrays (not serialization)
- Powerful **features**: backends, metadata handling, named axis, …

Awkward Array scales to large (high energy physics) analysis needs:

- **Parallelize** each analysis step **on a cluster** with dask (“Map Reduce”)
- IO-optimization: **load only what you need** from disk
  (reduces memory and IO overhead a lot)

Thank you very much for your attention!