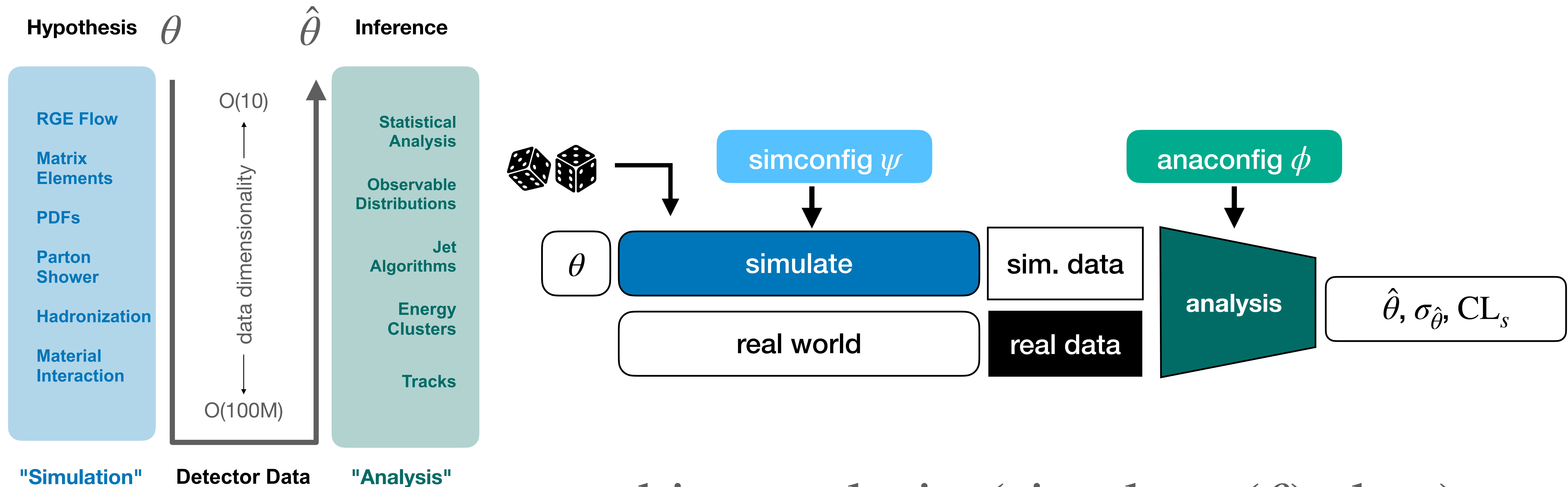
Discussion Points

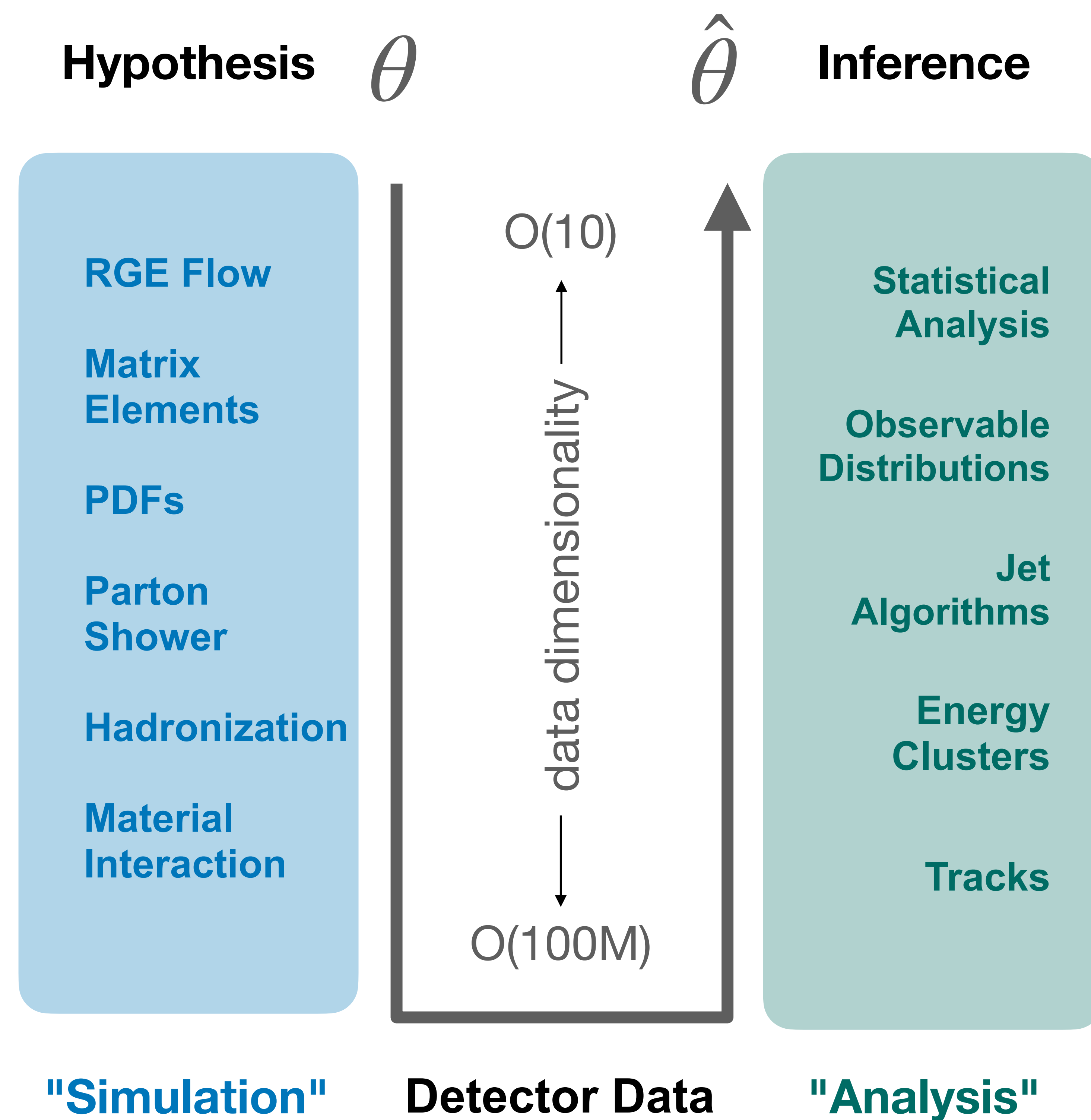
Differentiable Analyssis

Layout of Analysis

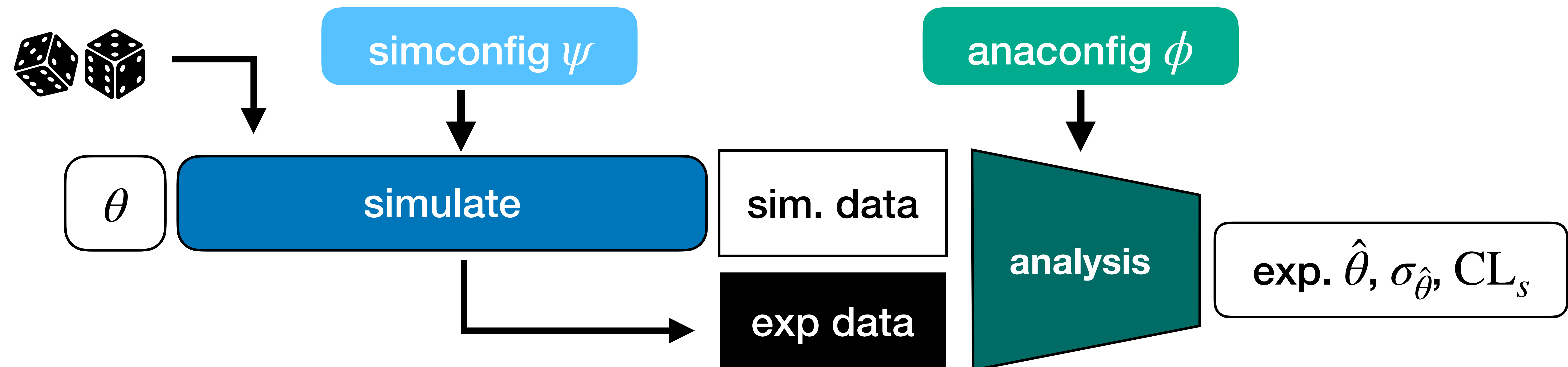


Nota bene: boundary between Simulate/Analysis can be unsharp (think: reco, ...)

Layout of Analysis



during analysis design, we use
expected/asimov data as proxy
still allowed to change configs ψ, ϕ

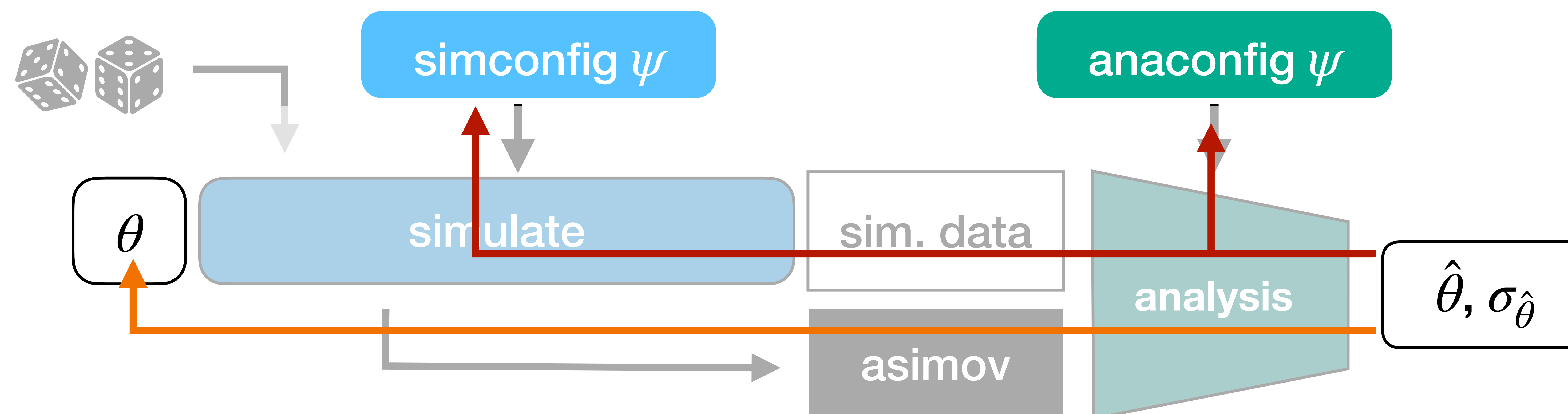


$$\mathrm{obj} = (\mathrm{analysis}_\phi \circ \mathrm{simulate}_\psi)(\theta)$$

Which Gradients

Two types of gradients are interesting in HEP processing chain

- gradients w.r.t. physics pars: quantify sensitivity (Fisher, CLs,...)
  - during design and at inference (i.e. on real data)
- gradients w.r.t. configuation: ability to optimize analysis
  - during design only



Remove stucture or Linalg on structured data

Differentiability lingo deals with flat vectors, i.e. functions of $f: \mathbb{R}^N \rightarrow \mathbb{R}^M$ giving rise to $M \times N$ Jacobians $\partial(\ldots)/\partial(\ldots)$

Got to represent data structure as flat data: we know how to do this!

- only data counts, everything else is structural "window-dressing"

```python
import awkward as ak
d = [{
    'weight': 1.02,
    'met': 75.23,
    'jets': [
        {'pt': 26.0, 'eta': -2.3, 'tracks': [
            {'pt': 14.0, 'eta': -2.5},
            {'pt': 7.0, 'eta': -2.3},
        ]},
        {'pt': 47.0, 'eta': 1.4, 'tracks': [
            {'pt': 27.0, 'eta': 1.5},
            {'pt': 17.0, 'eta': 1.7},
        ]}
    ],
}]
```

```python
event = ak.Array(d)
schema, length, buffers = ak.to_buffers(rec)
buffers
```

```
{'part0-node1-data': array([1.02]),
 'part0-node2-data': array([75.23]),
 'part0-node3-offsets': array([0, 2], dtype=int64),
 'part0-node5-data': array([26., 47.]),
 'part0-node6-data': array([-2.3,  1.4]),
 'part0-node7-offsets': array([0, 2, 4], dtype=int64),
 'part0-node9-data': array([14.,  7., 14.,  7.]),
 'part0-node10-data': array([-2.5, -2.3, -2.5, -2.3])}
```

```python
import numpy as np
np.concatenate([v for k,v in buffers.items() if '-data' in k])
```

```
array([ 1.02, 75.23, 26.  , 47.  , -2.3 ,  1.4 , 14.  ,  7.  , 14.  ,
        7.  , -2.5 , -2.3 , -2.5 , -2.3 ])
```

Here: this event is a vector in $\mathbb{R}^{14}$

NB: events w/ diff. multiplicity live in diff. spaces

Remove stucture or Linalg on structured data

Differentiability lingo deals with flat vectors, i.e. functions of $f: \mathbb{R}^N \to \mathbb{R}^M$ giving rise to $M \times N$ Jacobians $\partial(\ldots)/\partial(\ldots)$

Got to represent data structure as flat data: we know how to do this!

- only data ... ressing"

awkward array already nicely disentangles "structural" data
from "real"/floating data

it would be a key capability for awkward-array
to have "differentiable" (versions of)
`to_buffers` & `from_buffers`

from_buffers adopt buffers from JAX, turn into differentiable, stuctured ak.Arrays
to_buffers: turn structured data into flat JAX arrays for bulk processing

...ere: this event
...s a vector in $\mathbb{R}^{14}$

```
1  import awkward as ak
2  d = [{
3      'weight': 1.02,
4      'met': 75.23,
5      'jets': [
6          {'pt': 26.0,
7              {'pt': 1
8              {'pt': 7
9          ]},
10         {'pt': 47.0,
11             {'pt': 2
12             {'pt': 17.0, 'eta': 1.7}],
13         ]}
14     ],
15 }]
```

```
1  import numpy as np
2  np.concatenate([v for k,v in buffers.items() if '-data' in k])
```

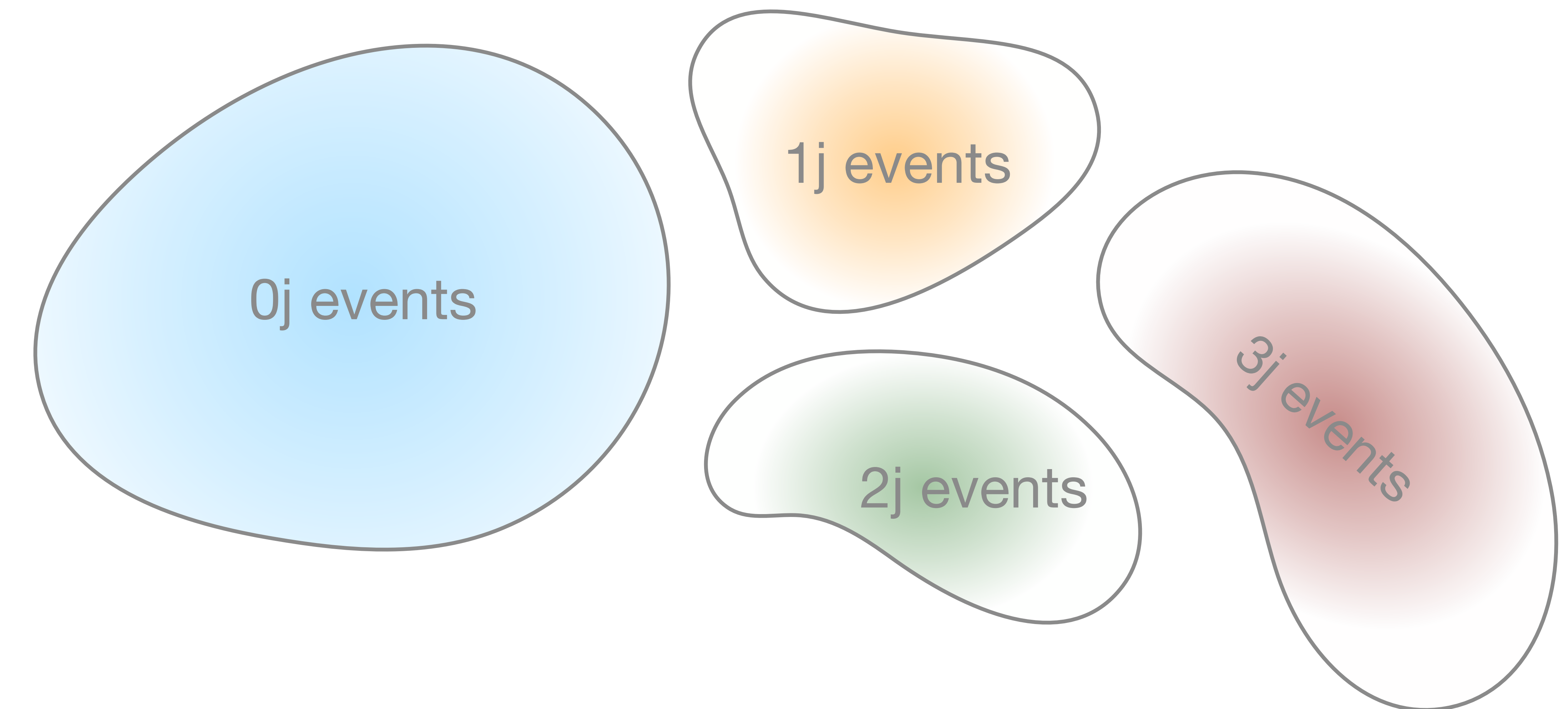
```
array([ 1.02, 75.23, 26.  , 47.  , -2.3 ,  1.4 , 14.  ,  7.  , 14.  ,
        7.  , -2.5 , -2.3 , -2.5 , -2.3 ])
```

NB: events w/ diff. multiplicity live in diff. spaces

How to connect Patches w/ diff. structure?

1 jet event

```
34  import awkward as ak
35  d = ak.Array([{
36      'weight': 1.02,
37      'met': 75.23,
38      'jets': [
39          {'pt': 26.0, 'eta': -2.3, 'tracks': [
40              {'pt': 14.0, 'eta': -2.5},
41              {'pt': 7.0, 'eta': -2.3},
42          ]}
43      ],
44  }])
45
```



1 jet event + 1 ghost jet

```
1   import awkward as ak
2   d = ak.Array([{
3       'weight': 1.02,
4       'met': 75.23,
5       'jets': [
6           {'pt': 26.0, 'eta': -2.3, 'tracks': [
7               {'pt': 14.0, 'eta': -2.5},
8               {'pt': 7.0, 'eta': -2.3},
9           ]},
10          {'pt': 0.0, 'eta': 0.0, 'tracks': [
11              {'pt': 0.0, 'eta': 0.0},
12              {'pt': 0.0, 'eta': 0.0},
13          ]}
14      ],
15  }])
```

2 jet event

```
1   import awkward as ak
2   d = [{
3       'weight': 1.02,
4       'met': 75.23,
5       'jets': [
6           {'pt': 26.0, 'eta': -2.3, 'tracks': [
7               {'pt': 14.0, 'eta': -2.5},
8               {'pt': 7.0, 'eta': -2.3},
9           ]},
10          {'pt': 47.0, 'eta': 1.4, 'tracks': [
11              {'pt': 27.0, 'eta': 1.5},
12              {'pt': 17.0, 'eta': 1.7},
13          ]}
14      ],
15  }]
```

How to connect Patches w/ diff. structure?

1. embedding smaller events in larger events (is there physics motivate embedding?)

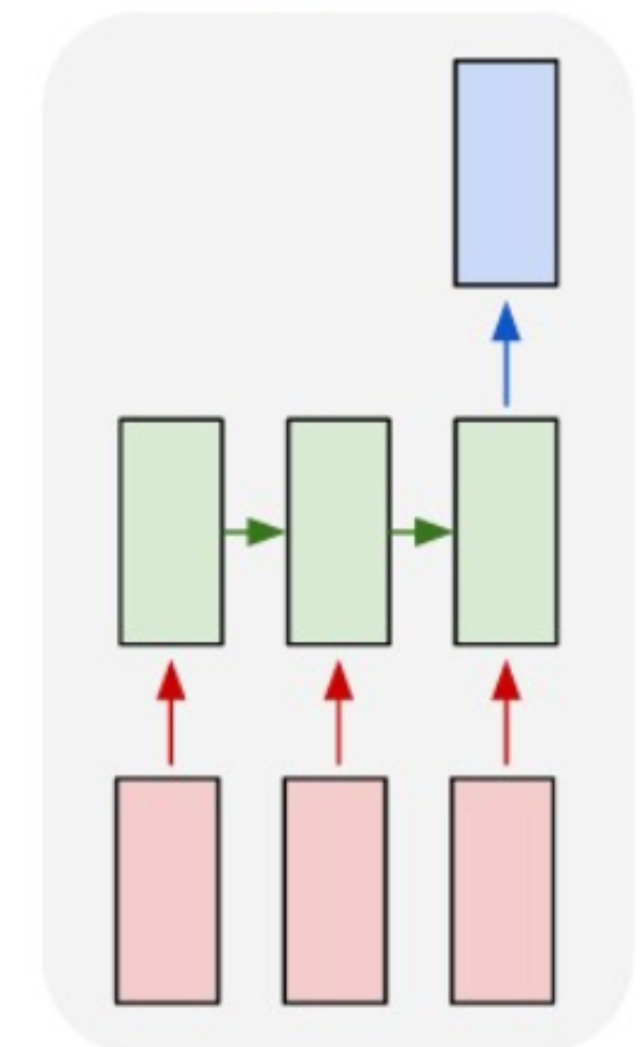
1 jet event

```
import awkward as ak
d = ak.Array([{
    'weight': 1.02,
    'met': 75.23,
    'jets': [
        {'pt': 26.0, 'eta': -2.3, 'tracks': [
            {'pt': 14.0, 'eta': -2.5},
            {'pt': 7.0, 'eta': -2.3},
        ]}
    ],
}])
```

1 jet event + 1 ghost jet: 2 jet event

```
import awkward as ak
d = ak.Array([{
    'weight': 1.02,
    'met': 75.23,
    'jets': [
        {'pt': 26.0, 'eta': -2.3, 'tracks': [
            {'pt': 14.0, 'eta': -2.5},
            {'pt': 7.0, 'eta': -2.3},
        ]},
        {'pt': 0.0, 'eta': 0.0, 'tracks': [
            {'pt': 0.0, 'eta': 0.0},
            {'pt': 0.0, 'eta': 0.0},
        ]}
    ],
}])
```

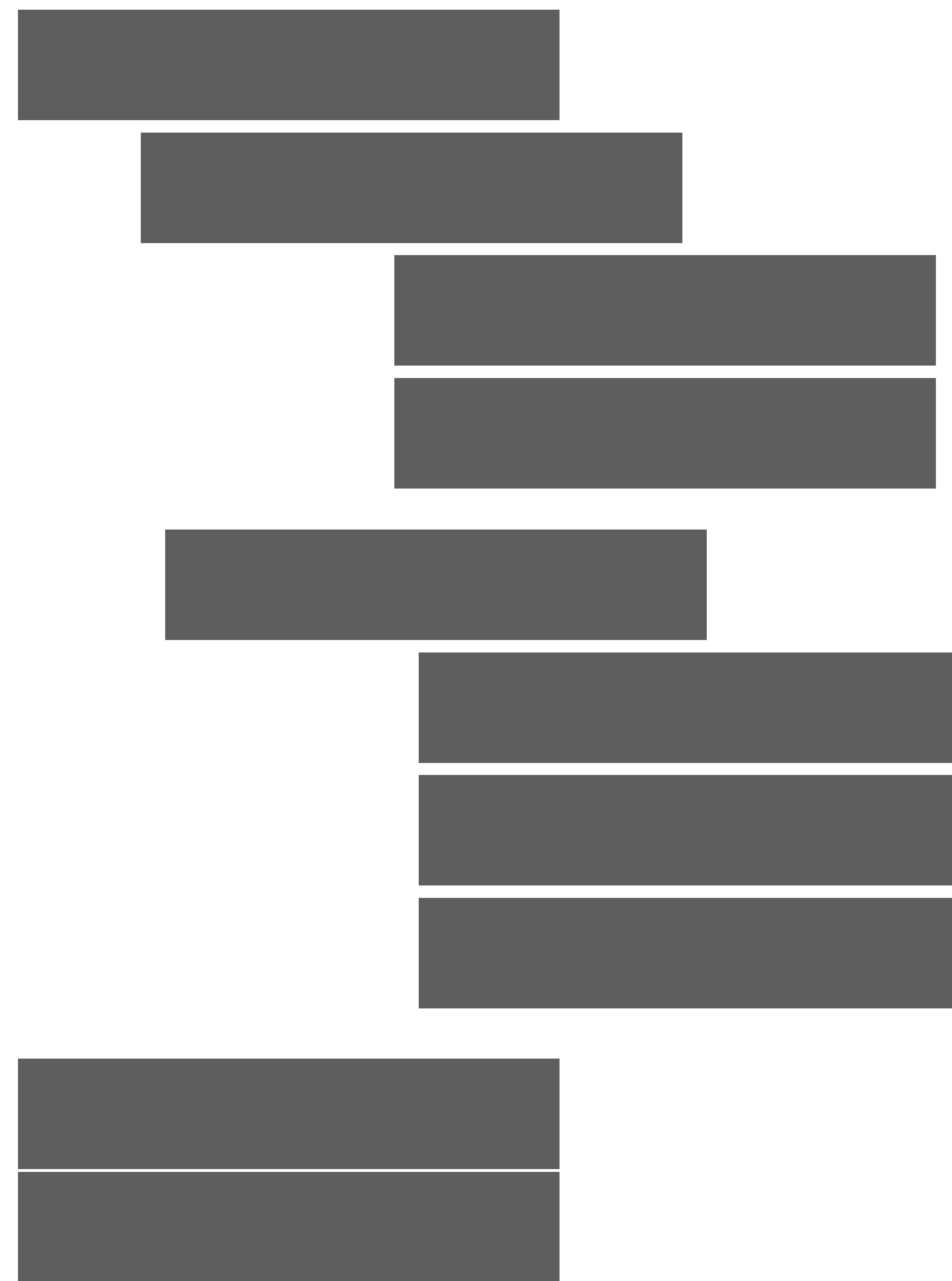
2. recursively update fixed-size repr w/ looping over

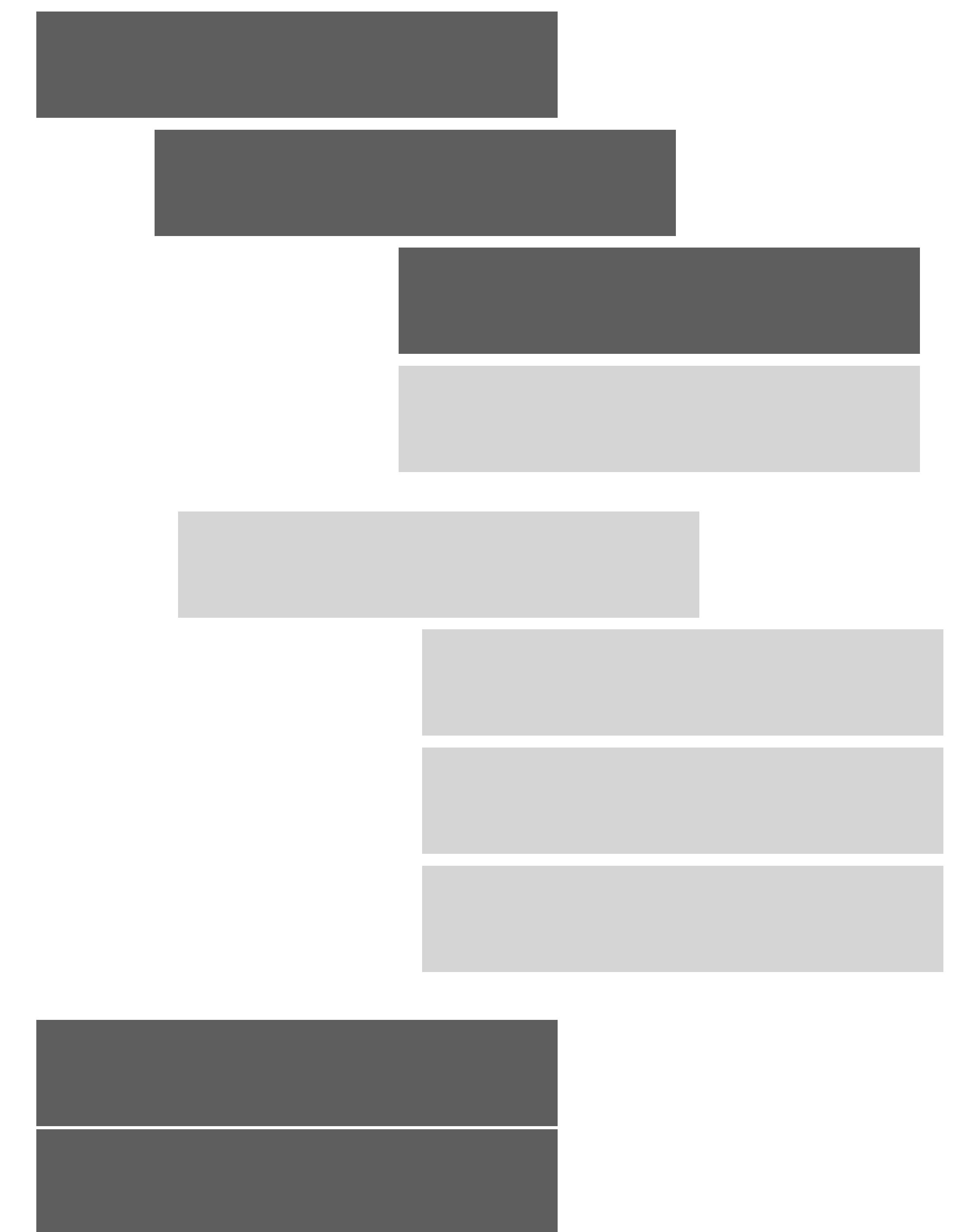


fixed-size data can still be physics-structured (see e.q. QCD-aware jets paper) (e.g. think: N-jettines, like N-subjettiness)

How to connect Patches w/ diff. structure?

Embedding nested to flat



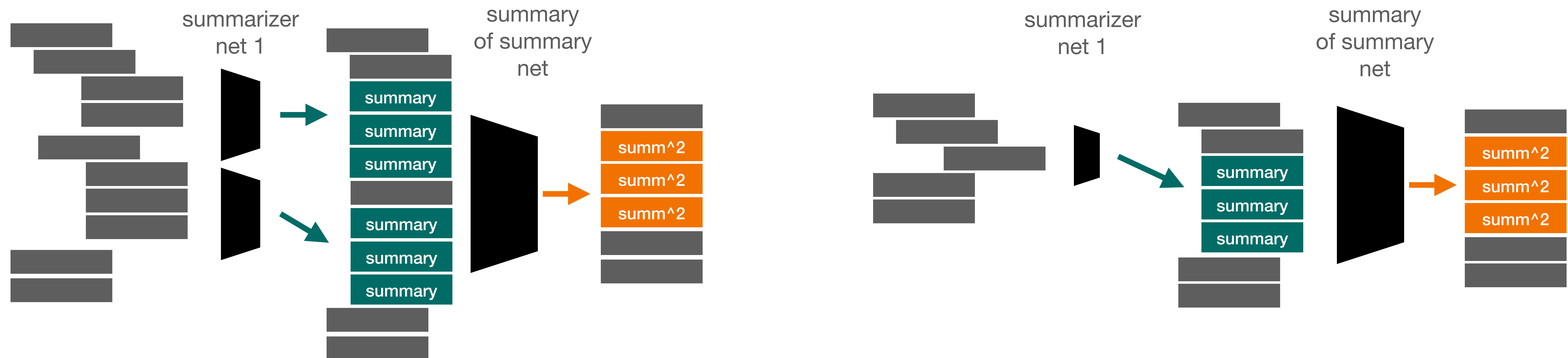


How to connect Patches w/ diff. structure?

Embedding nested to flat using recurrent architectures
think:

- "given tracks of jet summarize in 3 floats what's happening in the track-wise in this jet"
- "given jets (and their track summaries) summarize what's happening jet-wise

Result is flat non-nested structure, some of the summary variables likely to encode the strucure in a differentiable way



How to connect Patches w/ diff. structure?

Embedding
think:
- "given tr
- "given je

is jet"

Result is fl
the strucur

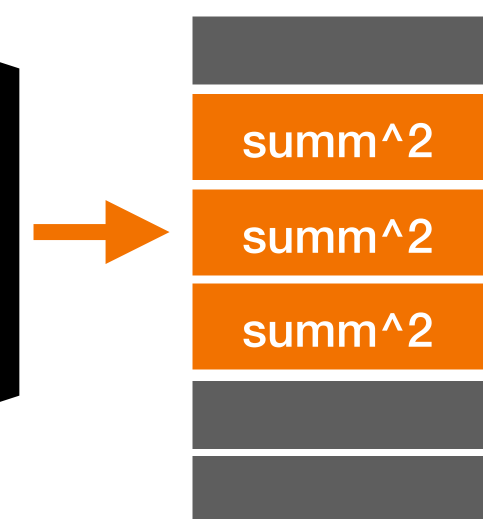
summ
ne

Building recursive neural embedders (encoders) seems like a nice project
(let me kow if you wanna do it, happy to collab)

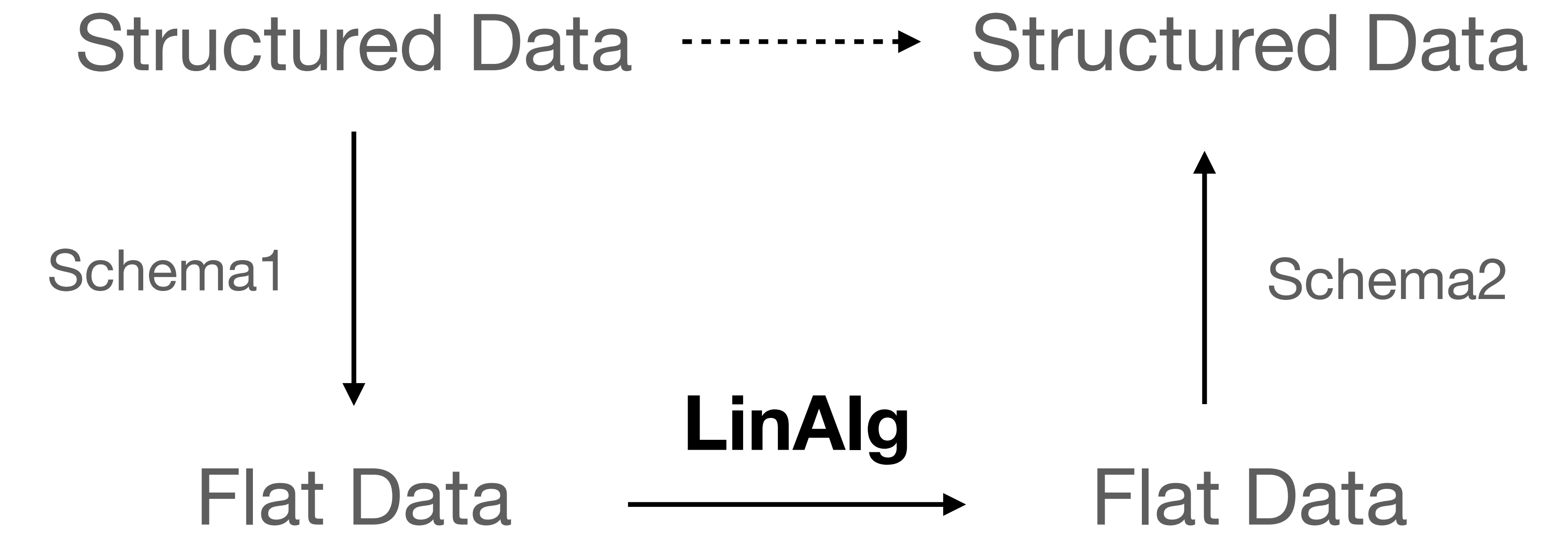
```
import neurawk as nk
import awkward as ak
events = ak.from_parquet('data.pq')
embedder = nk.Embedder(ak.type(events), opts = ...)
flatevents = embedder.flatten(events)
```

lowish-hanging, continuously improvable fruit?
community activity that can grow around schemas & Open Data
(CMS NanoAOD, ATLAS PHYSLITE, HepMC, LHE, ...)

tldr ways around shape variability will ignore
shape-discrepancies for rest of slides

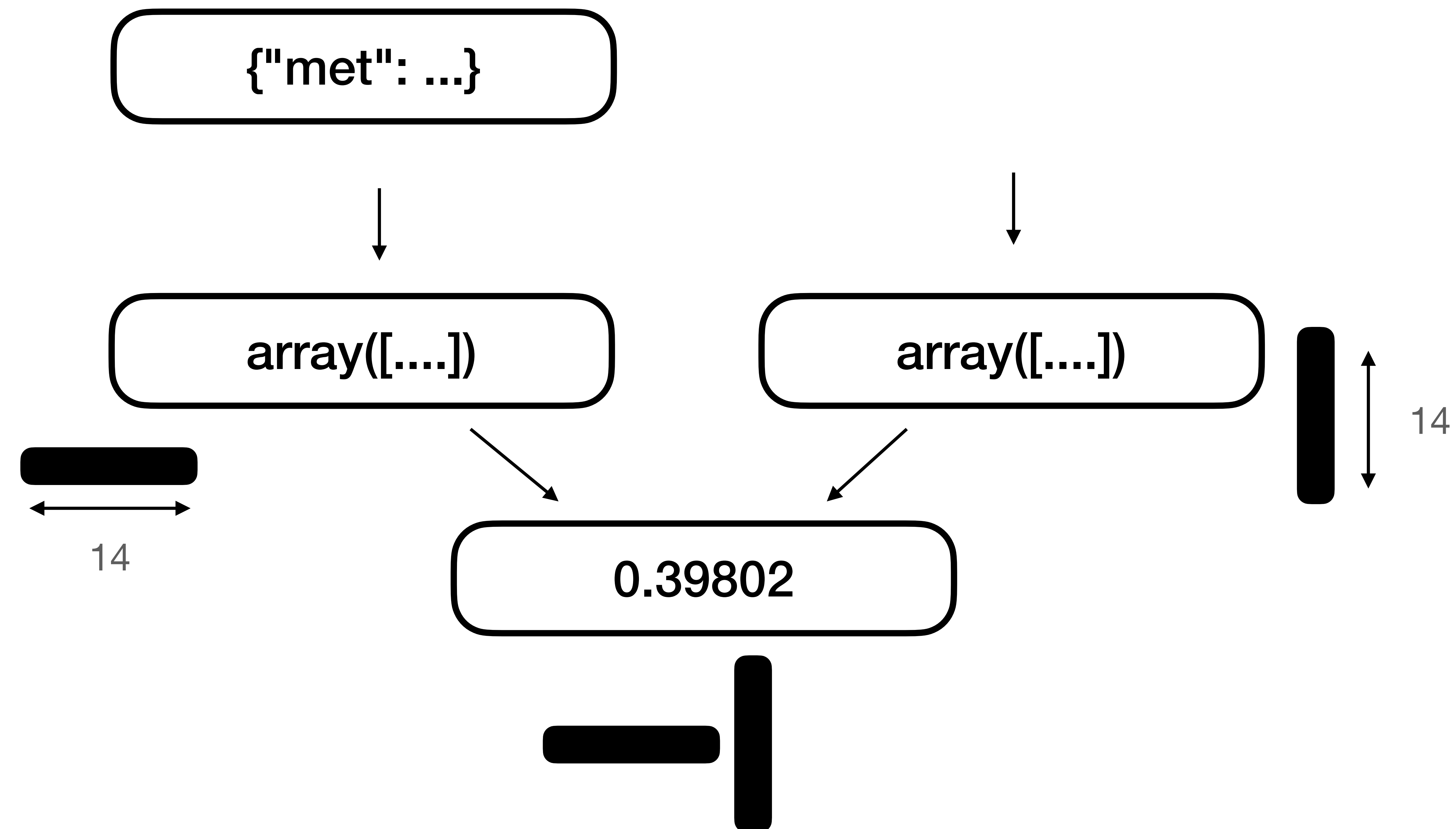


Remove stucture or Linalg on structured data

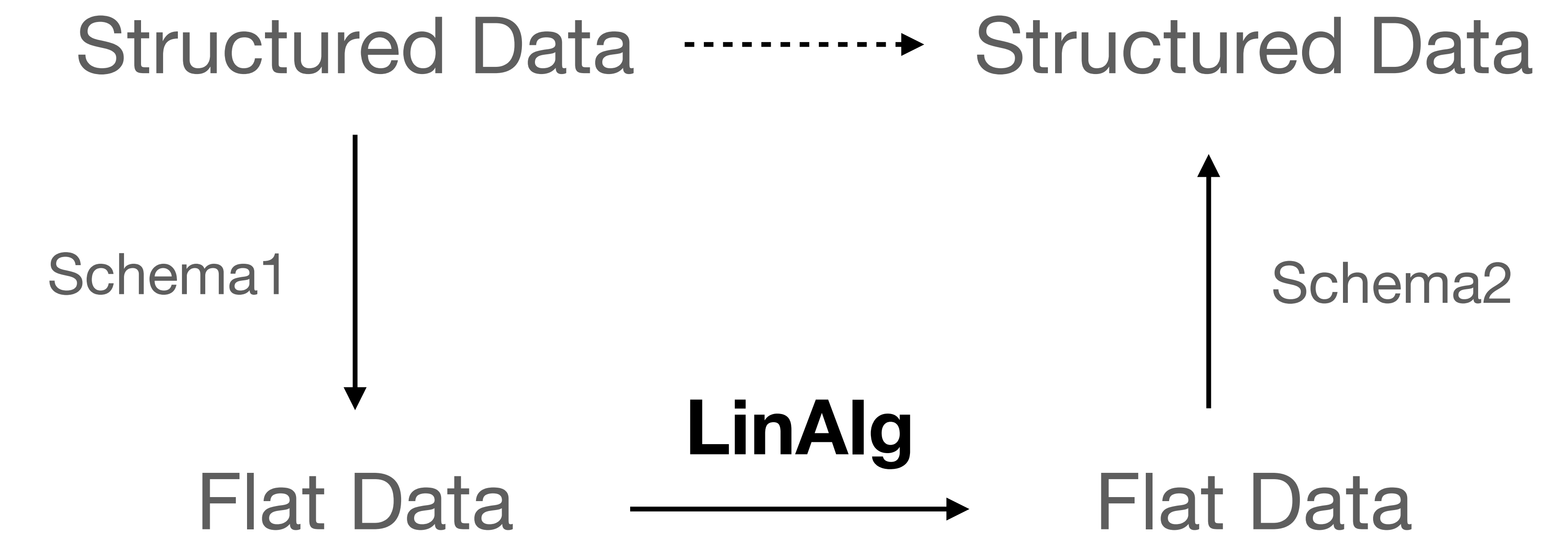


Can do linear algebra by on structured data by going through flatten/unflatten

e.g. innner product in event space



Remove stucture or Linalg on structured data



Alternatively: could consistently define LinAlg operations directly on structured data

- depends how many meaningful operations exist for us
- norm, inner product, matmul, inv(?), ...

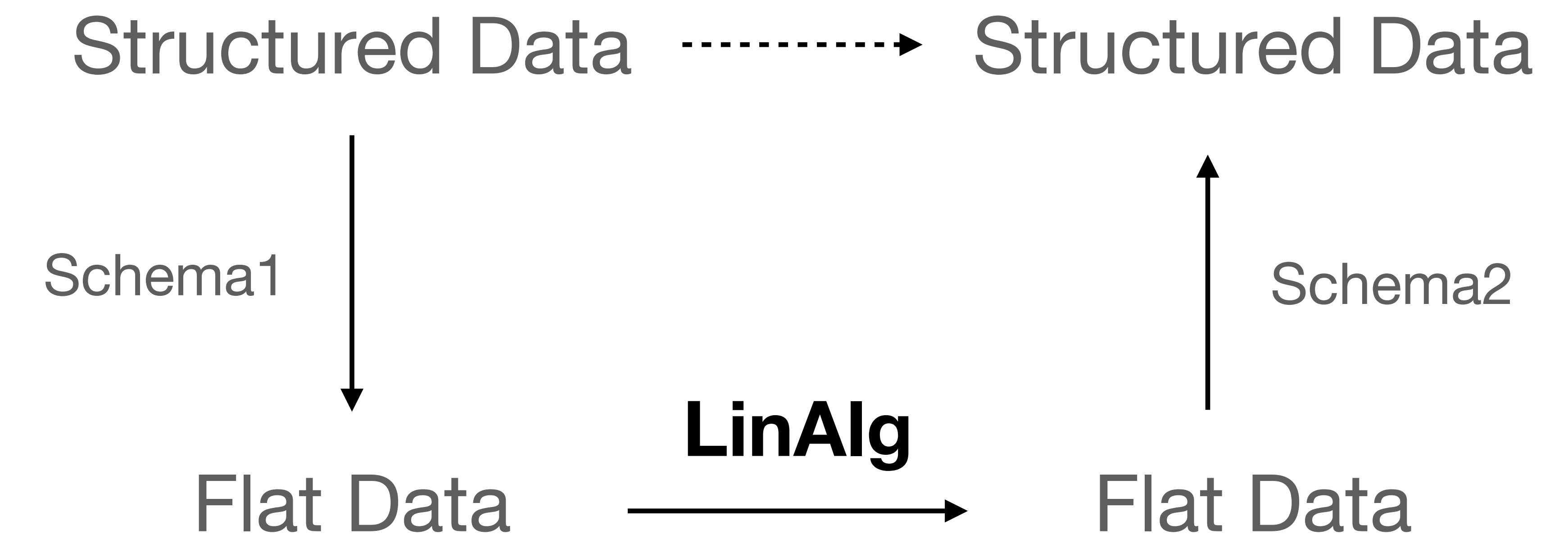
$$e_1 \cdot e_2 = \sum_{\mathrm{l \in leaf}} e_1^l e_2^l = e_1^{\mathrm{met}} e_2^{\mathrm{met}} + e_1^{\mathrm{weight}} e_2^{\mathrm{weight}} + e_1^{\mathrm{j1pT}} e_2^{\mathrm{j1pT}} + + e_1^{\mathrm{j1}\eta} e_2^{\mathrm{j1}\eta} + \ldots$$

$$x = Ae = \sum_{\mathrm{o \in outleaf, k \in inleaf}} A^{ok} e^k$$

schema: cartesian product of input & output schema

```
ak.linalg.dot(e1, e2) ?
ak.linalg.matmul(A,e) ?
```

Remove stucture or Linalg on structured data



Can do linear algebra by on structured data by going through flatten/unflatten

observables: vector-valued function of event

- matrices in flat space (if linear)

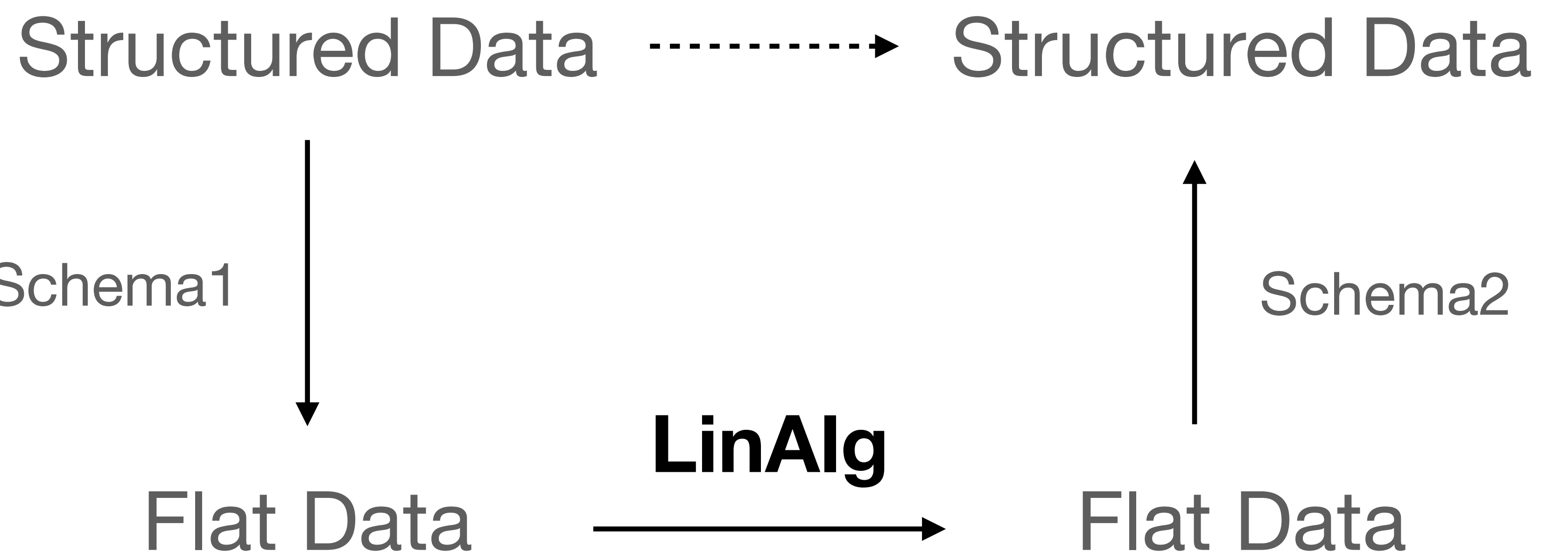
```
def ak_linear_program(e):
    return ak.Array({'a': 2*e.met, 'b': e.met + e.jets[0].pt})

print(ak_linear_program(event))
```

```
[{a: 150, b: 101}]
```

```
def ak_non_linear_program(e):
    return ak.Array({'a': np.sqrt(e.met), 'b': e.met / e.jets[0].pt})

print(ak_non_linear_program(event))
```

```
[{a: 8.67, b: 2.89}]
```

Remove stucture or Linalg on structured data



Can do linear algebra by on structured data ... atten

obse ... n of event

- ma

NB: for linear functions program representation is much more compact than matrix repr.
automatic sparsenes

also programs:
unified view of linear & non-linear funcs

```
1 def ak_linear_program(e):
2     return ak.Array({'a': 2*e.met, 'b': e.met + e.jets[0].pt})
3
4 print(ak_linear_program(event))
```

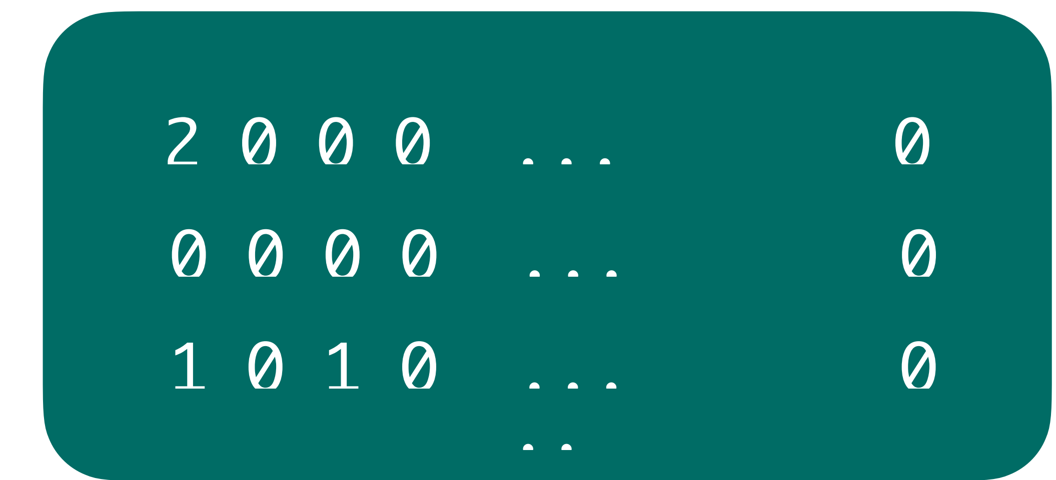
```
[{a: 150, b: 101}]
```

```
1 def ak_non_linear_program(e):
2     return ak.Array({'a': np.sqrt(e.met), 'b': e.met / e.jets[0].pt})
3
4 print(ak_non_linear_program(event))
```

```
[{a: 8.67, b: 2.89}]
```

a = 2*met

b = j1pt+met



met

weight

jet1pT

...

...

a = 2*met

b = j1pt+met

met

weight

jet1pT

...

Autodiff Recap:

Given (in general non-linear) $f : \mathbb{R}^N \to \mathbb{R}^M$, $y = f(x)$
we want to have gradients, e.g. Jacobian Matrix (linear by definition)

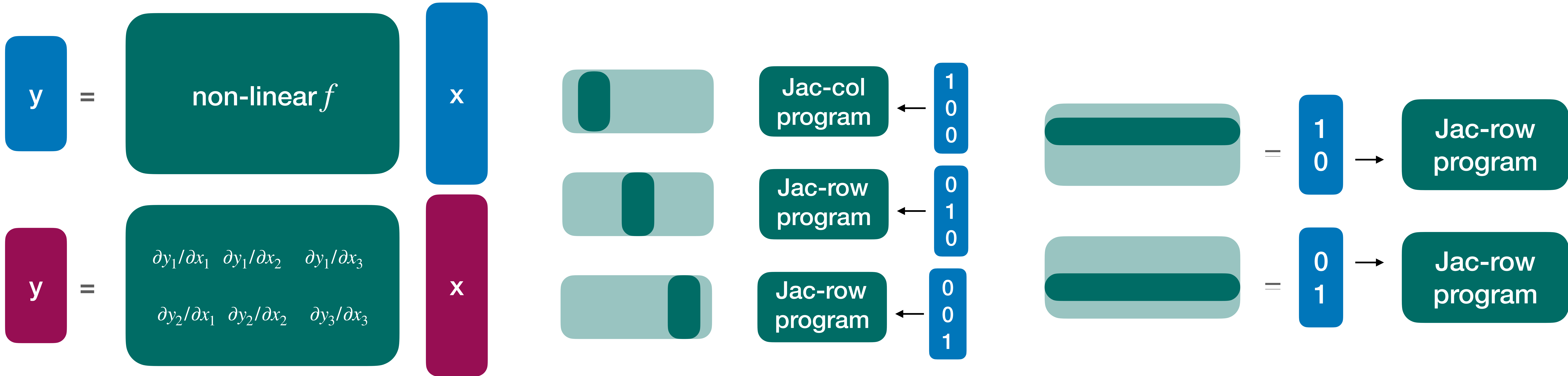
We've seen programs represent (linear & non-linear) functions in high-D much more efficiently than explicit representation (e.g. matrix)

Autodiff: given a program f represent Jac. Matrix as a (linear) program

- program either yields rows or columns of Jacbian

Autodiff: given a program f represent Jac. Matrix as a (linear) program

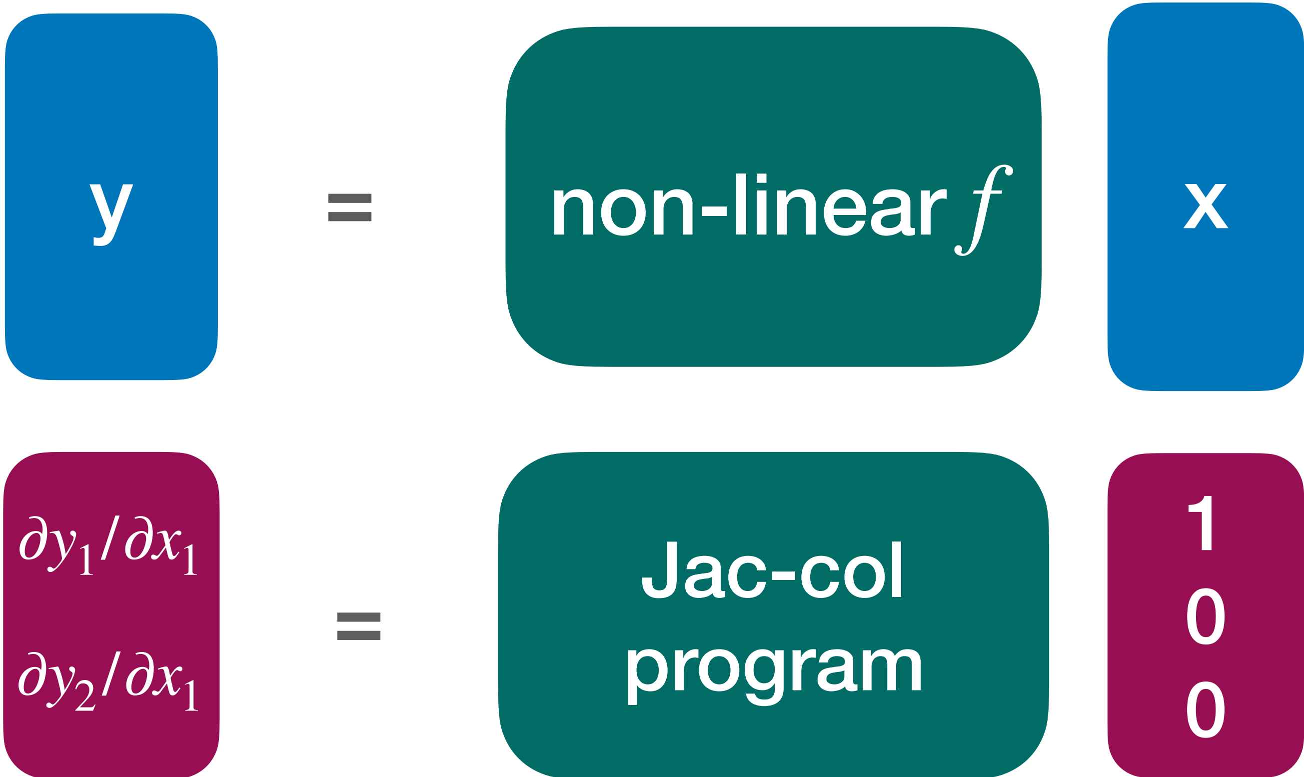
- program either yields rows (backprop) or columns (fwdprop)
- **inputs** to JacProg: same shape as **inputs** (fwd) **outputs** (bwd) of f
- **outputs** of JacProg: same shape of **output** (fwd) **input** (bwd) of f



Since shapes match in forward autodiff:
evaluate main program and JacProg in one go

"gradient" data gets produced sychronously
with the main output data

NB: this can be even at the file level



$$y = f(\theta) \rightarrow y, \bar{y} = f(\theta, \bar{\theta})$$

```
./program --in in.root --out out.parquet
./program --in in.root --ingrad ingrd.root
--out out.parquet --outgrad outgrd.parquet
```

For backward, cannot do it 'on-the-fly', but forward function now rerturns function w/ "inverse" signature (out → in)

gradient part gets produced asynchronously, requires book-keeping

NB: this can be even at the file level

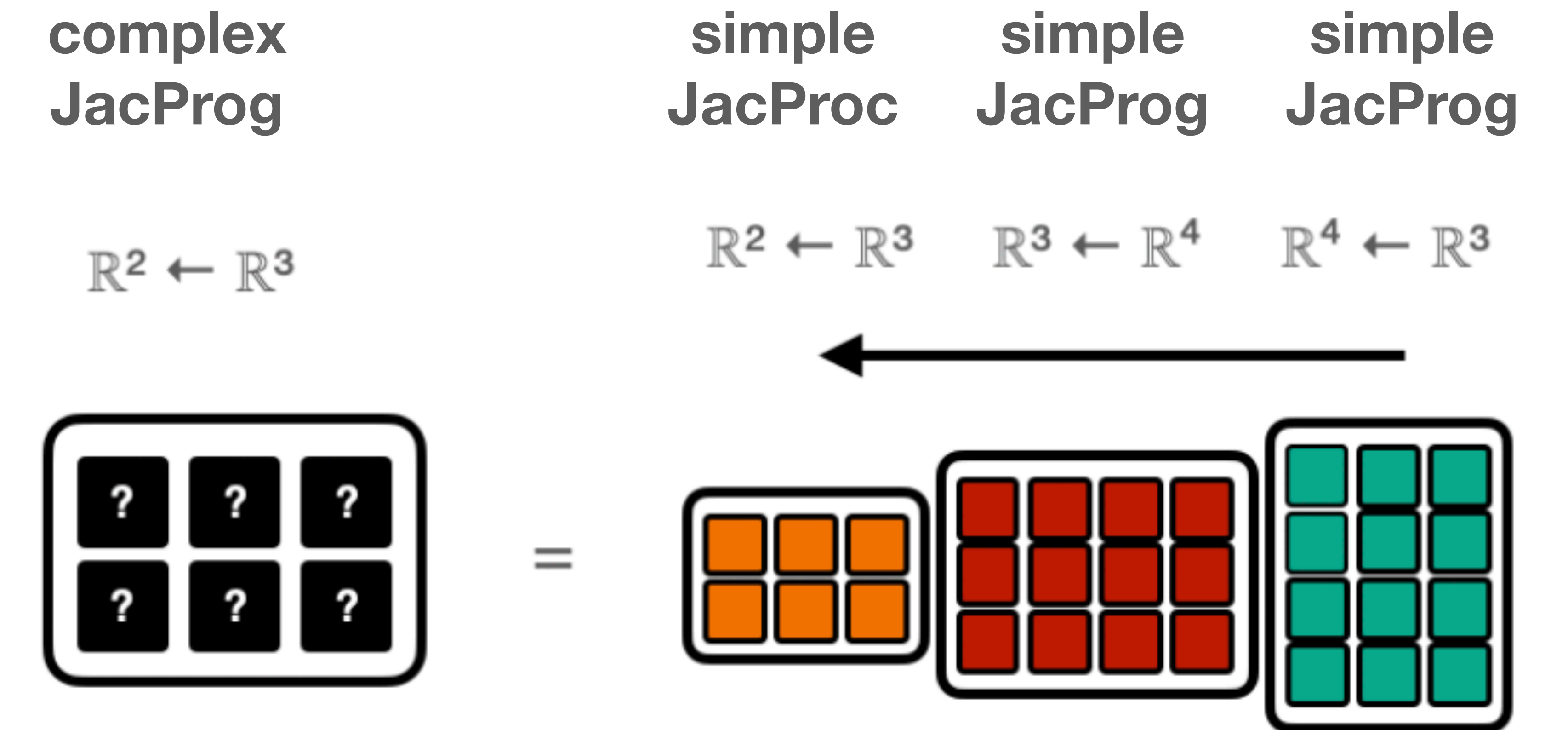
$$y = f(\theta) \rightarrow y, f_{\text{back}} = f(\theta) \qquad \bar{\theta} = f_{\text{back}}(\bar{y})$$

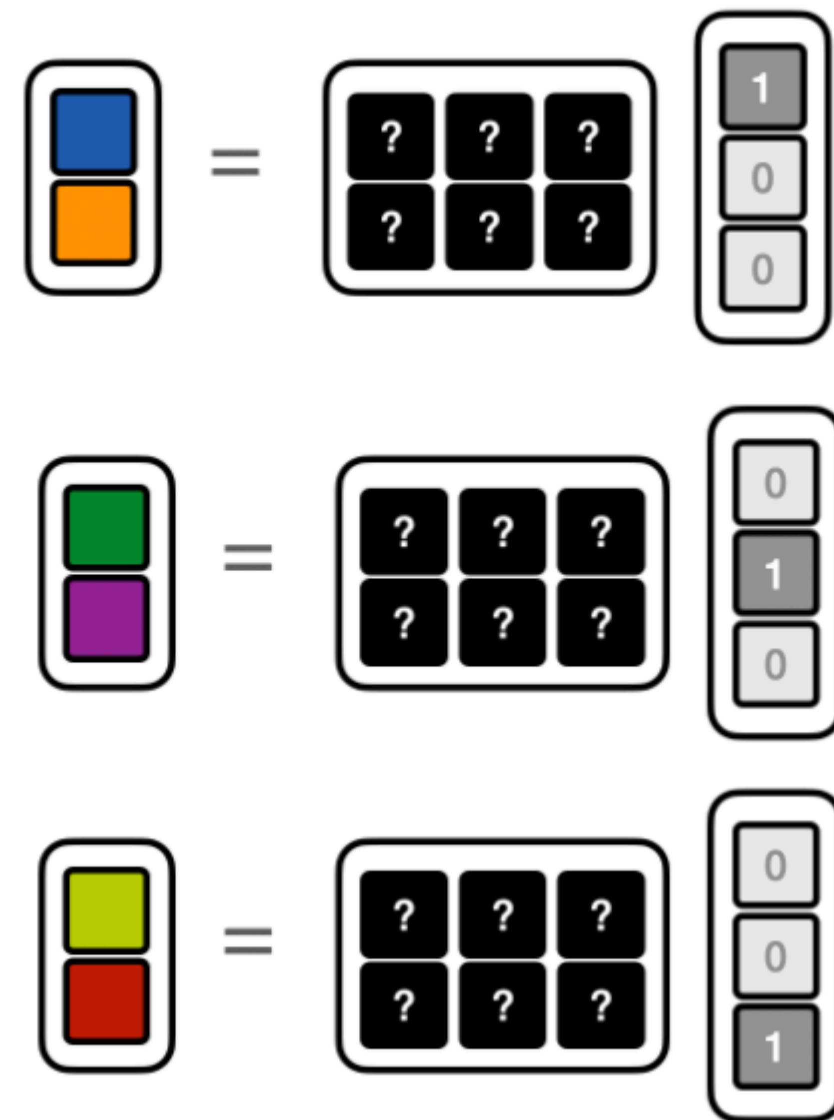
```
./program --in in.root --out out.parquet

./program --in in.root --out out.parquet
          --saveforback back.cfg

./backward --cfg back.cfg
--outgrad outgrd.parquet --ingrad ingrad.root
```

Composition

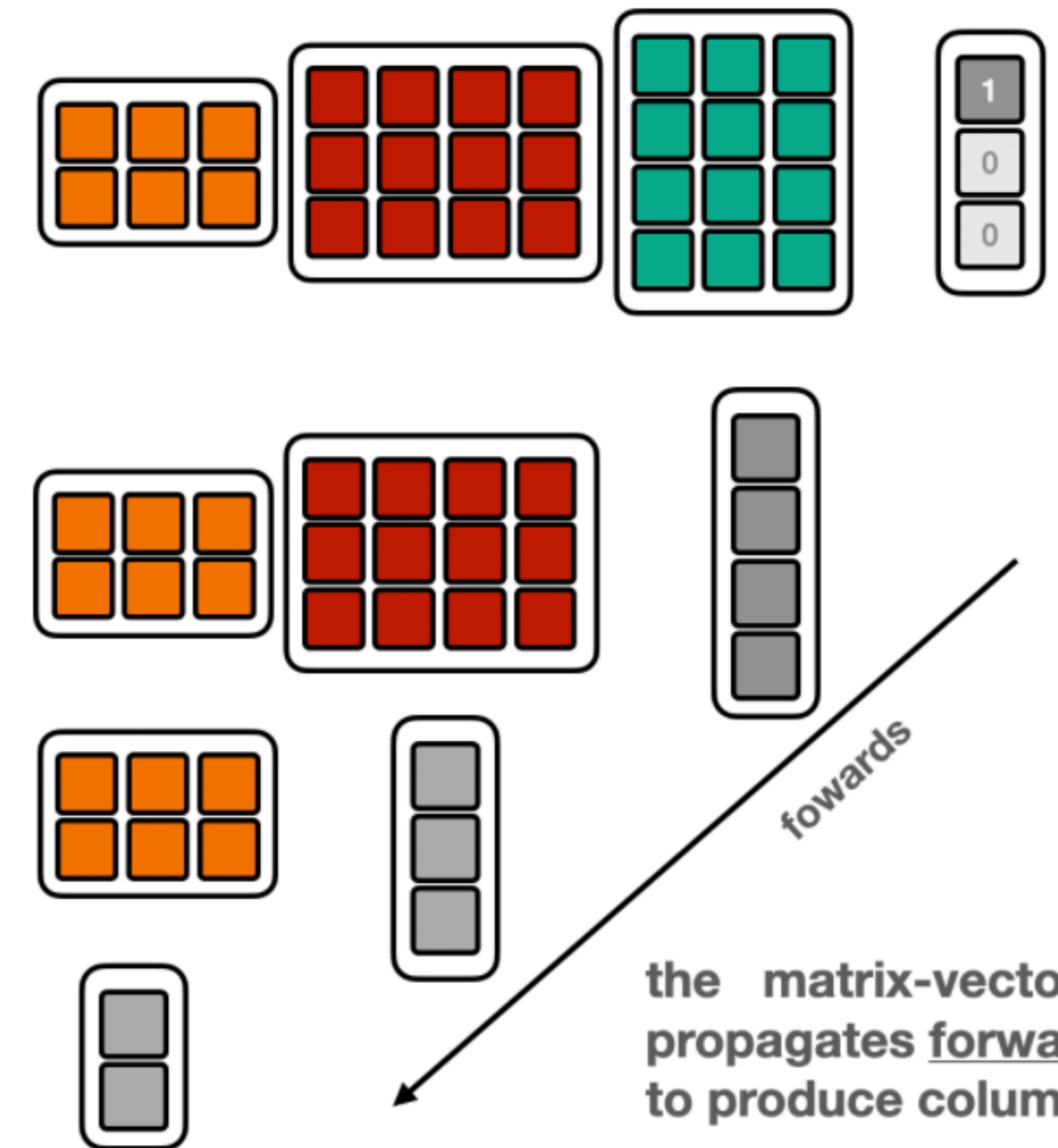
If we can get JacProgs for simple f

We can get JacProgs for complex f through composition (core of autodiff)





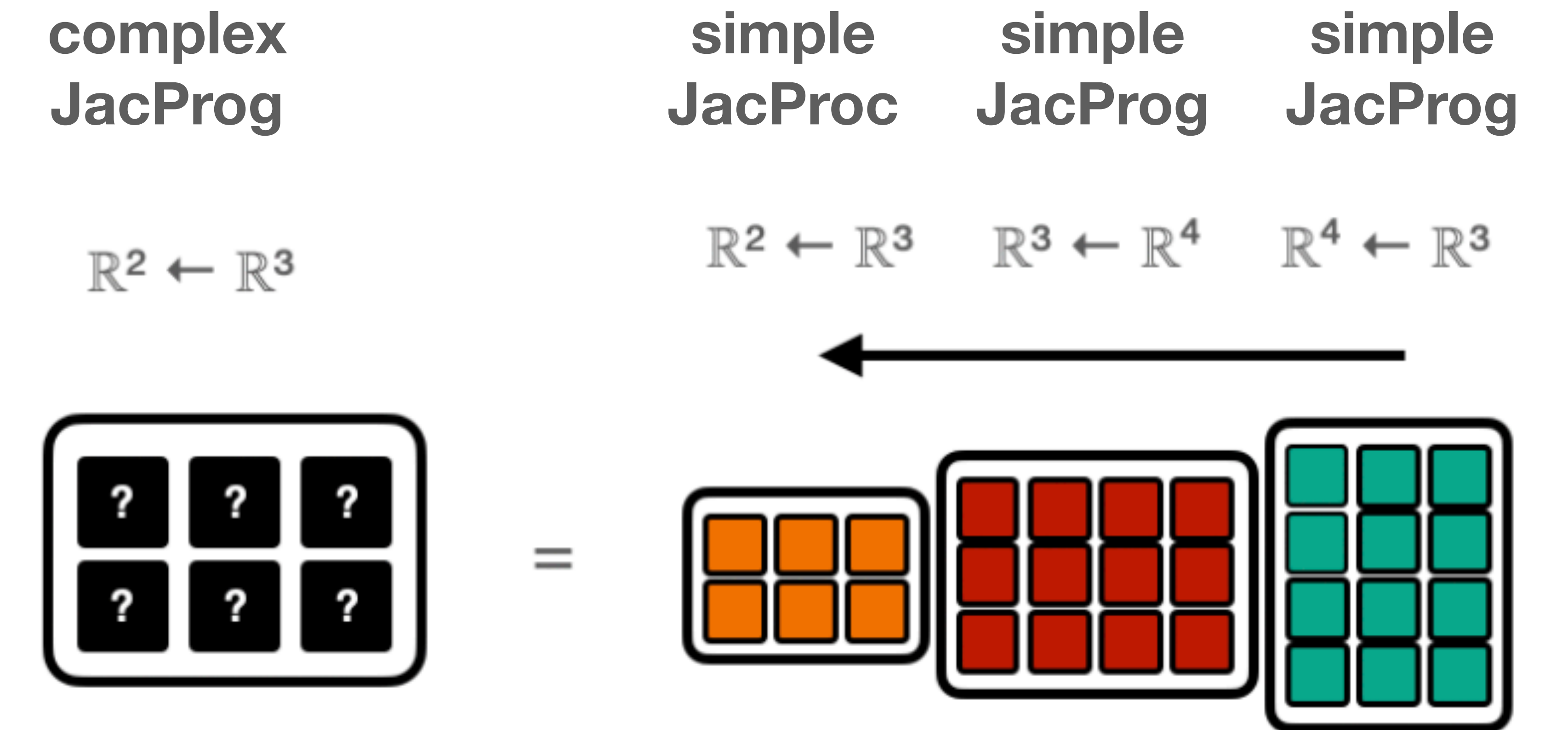
Linear Transformations: $\mathbb{R}^m \rightarrow \mathbb{R}^n$ can be fully characterized by Matrix-Vector Products

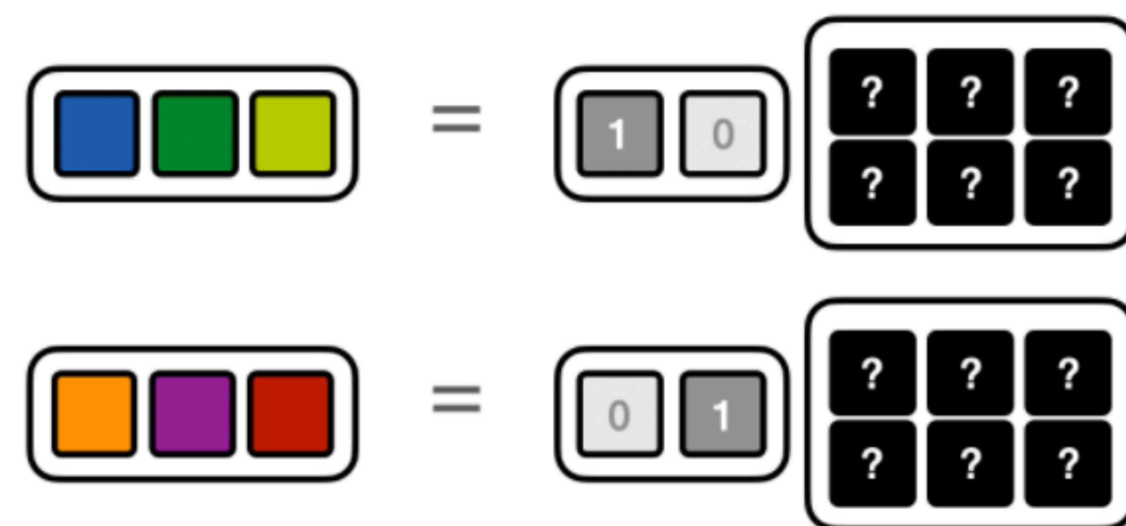


Composition

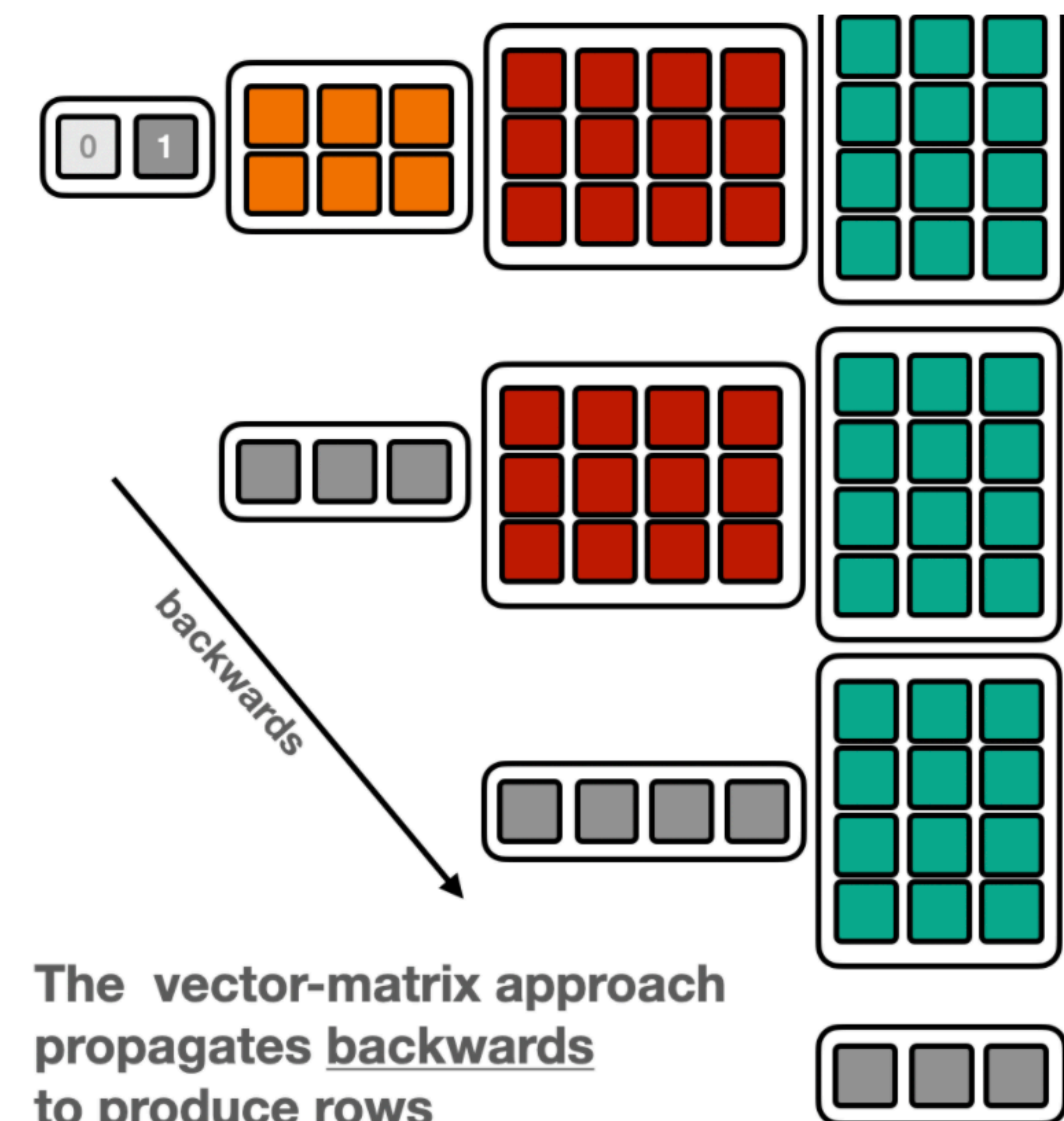
If we can get JacProgs for simple f

We can get JacProgs for complex f through composition (core of autodiff)





Linear Transformations: $\mathbb{R}^m \to \mathbb{R}^n$ can be fully characterized by Vector-Matrix Products, too.



The vector-matrix approach propagates backwards to produce rows

Forward

Backward

cfg.json

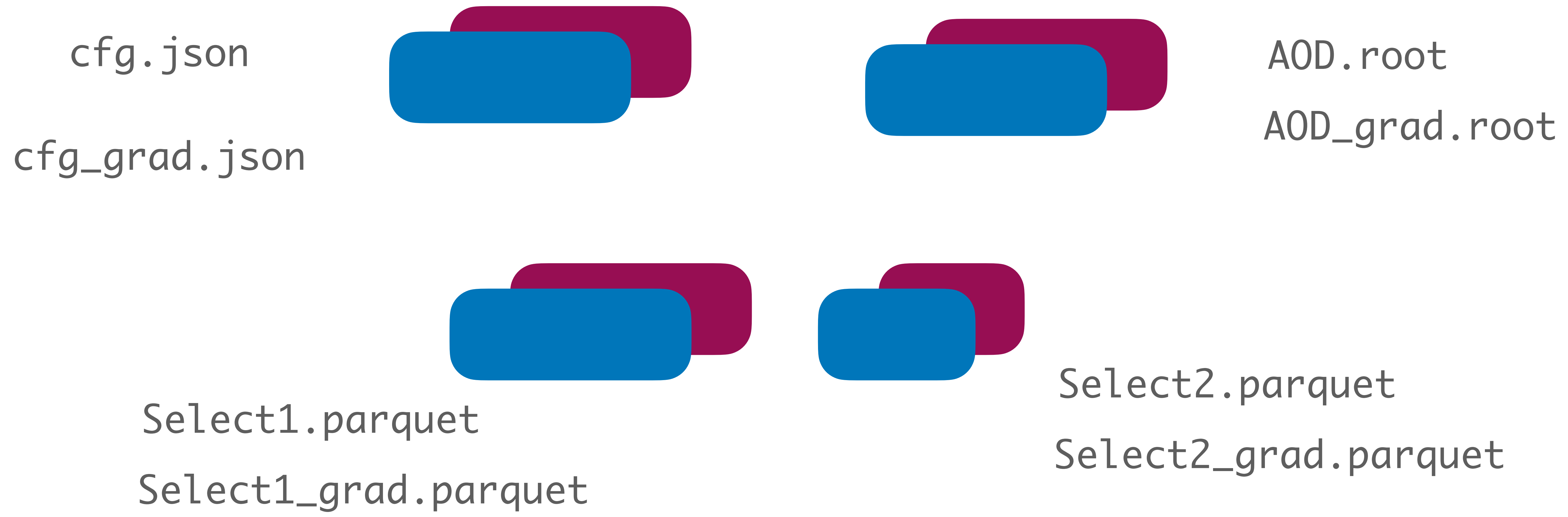
cfg_grad.json

AOD.root

AOD_grad.root

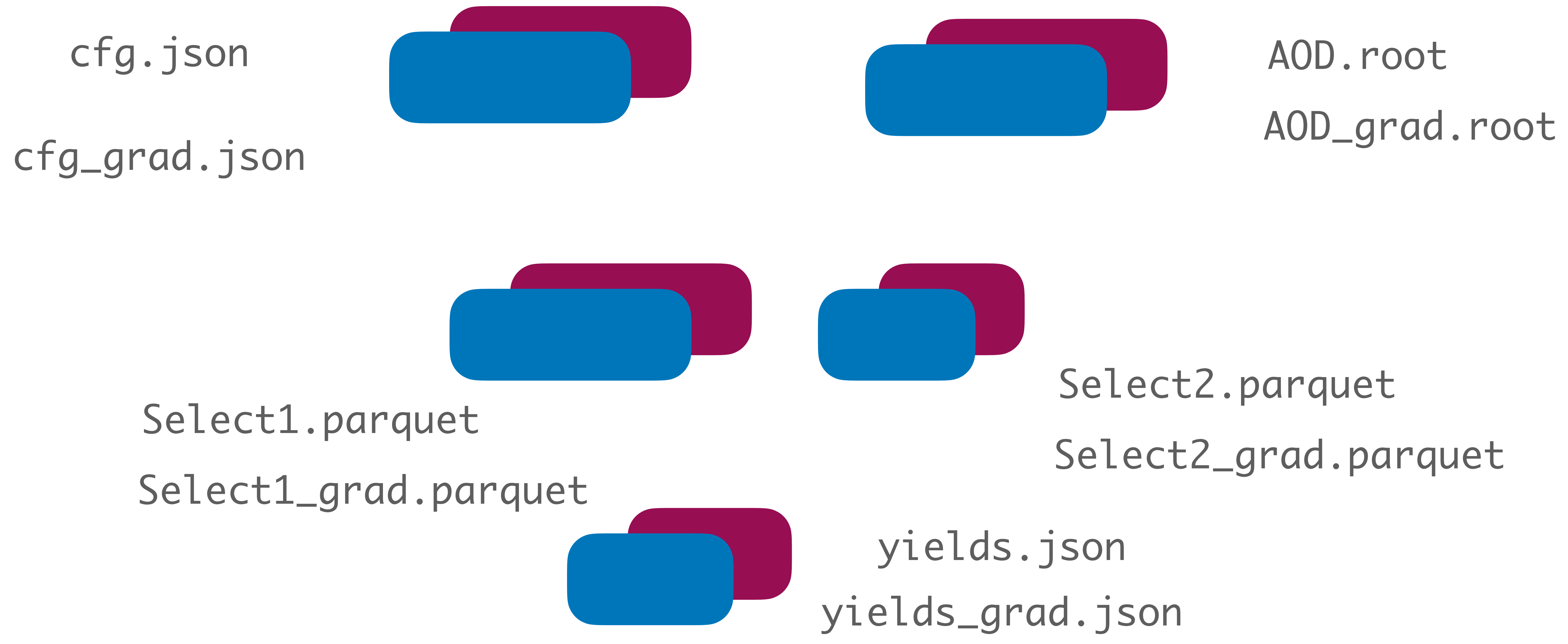
Forward

Backward



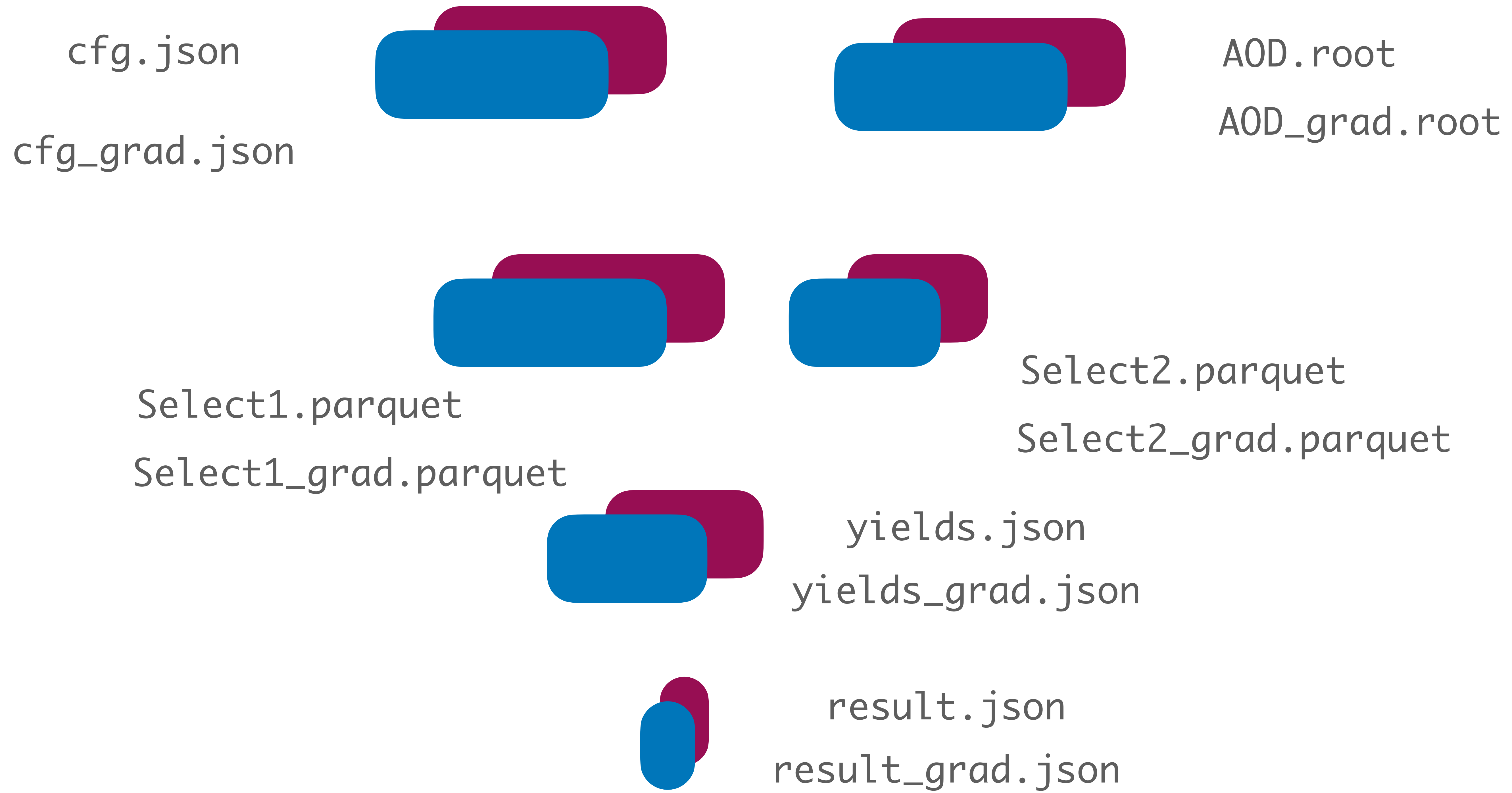
Forward

Backward



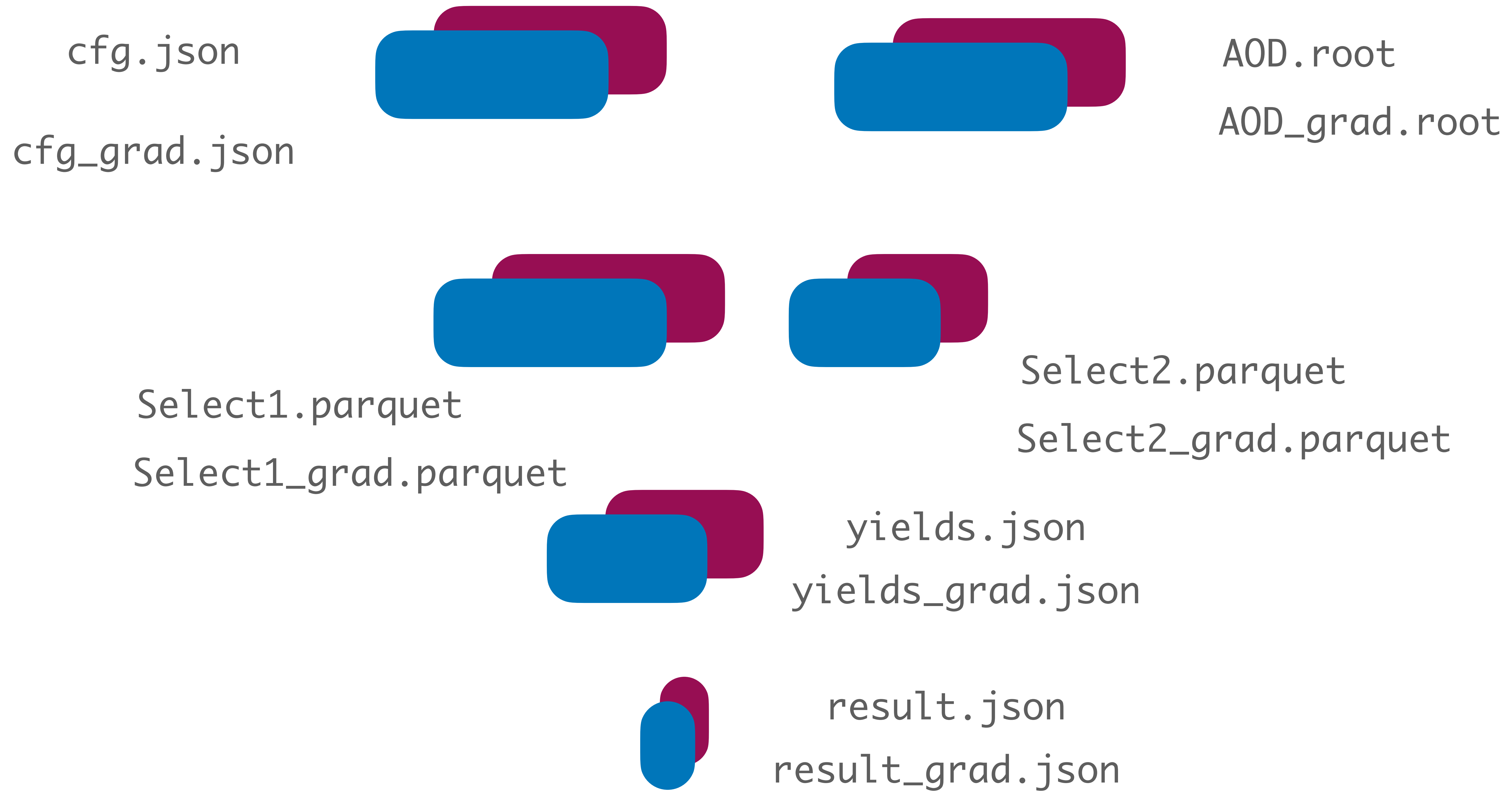
Forward

Backward



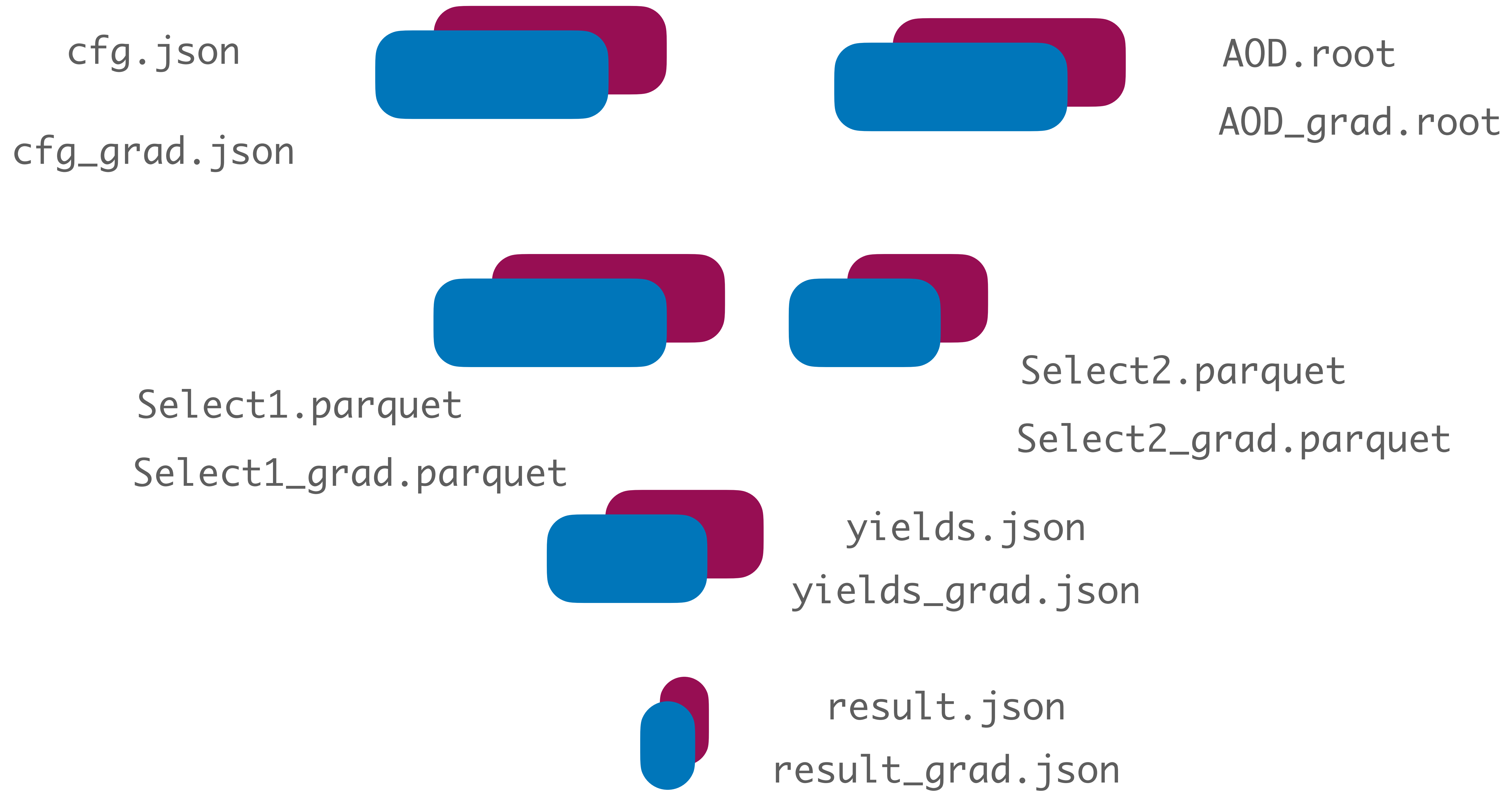
Forward

Backward



Forward

Backward



Forward

cfg.json

cfg_grad.json

AOD.root

AOD_grad.root

Select1.parquet

Select1_grad.parquet

Select2.parquet

Select2_grad.parquet

yields.json

yields_grad.json

result.json

result_grad.json

Backward

AOD.root

cfg.json

Forward

Backward

Forward

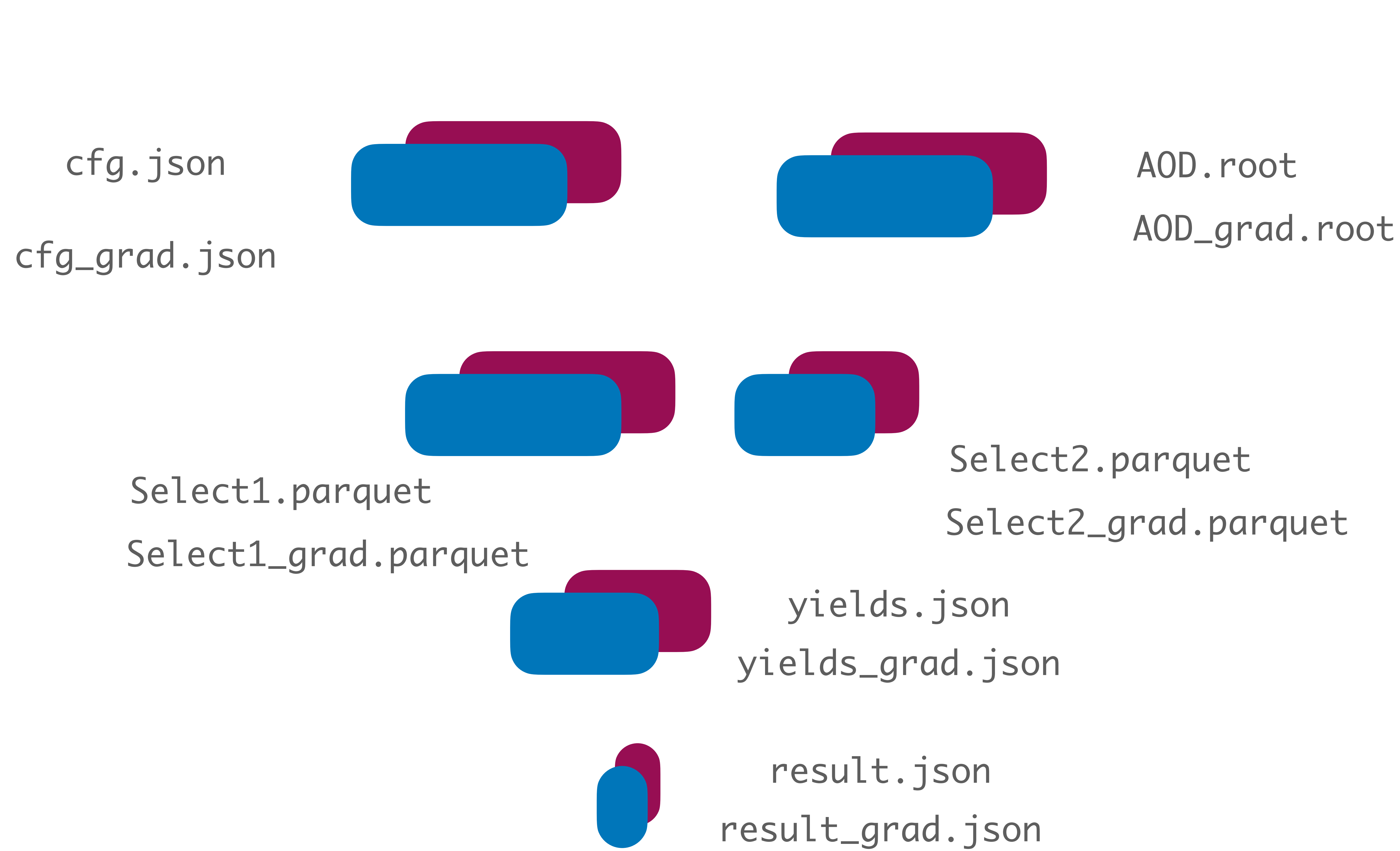
Backward

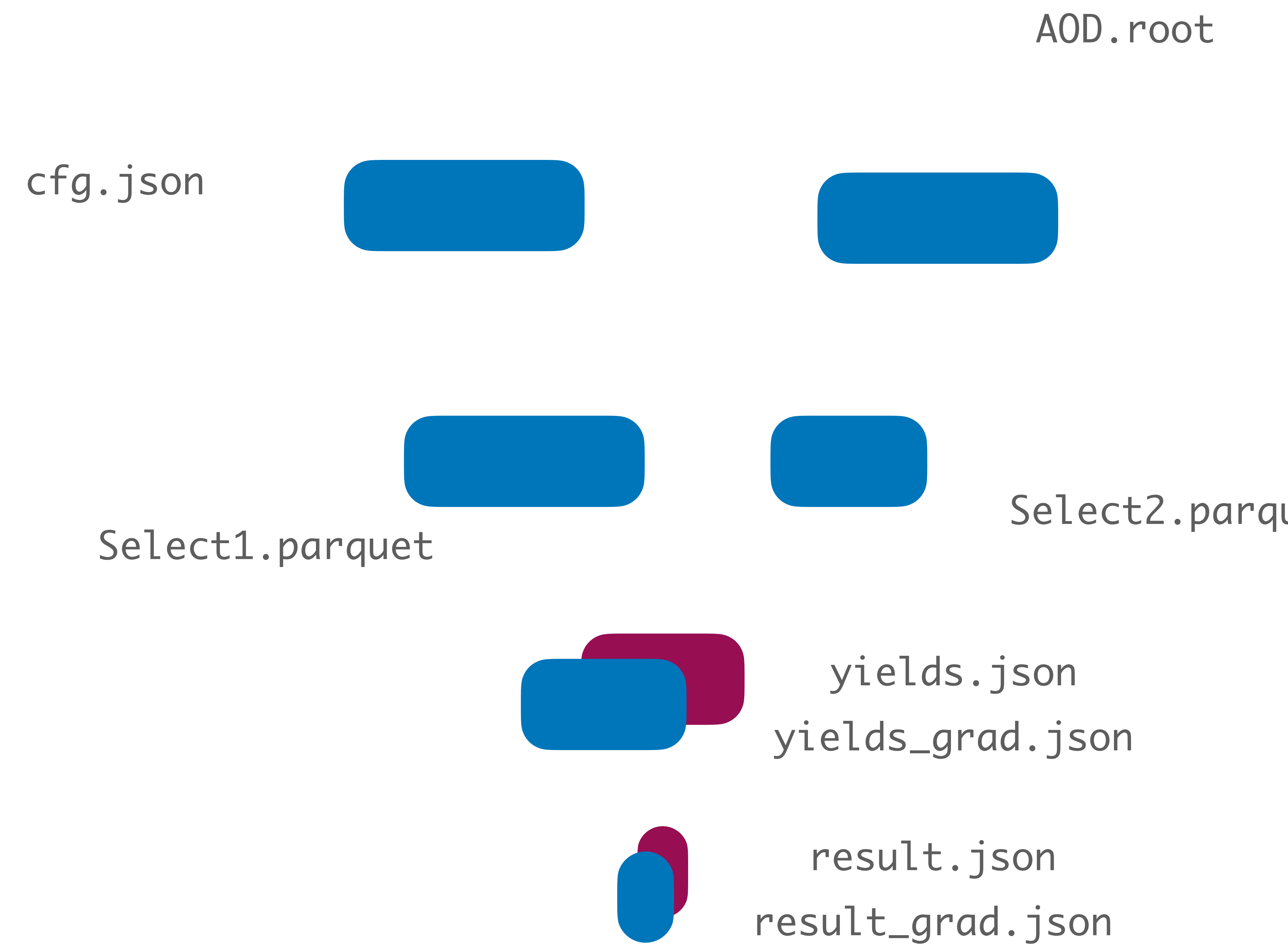
Forward

Backward

Forward

Backward


Forward
cfg.json
cfg_grad.json
AOD.root
AOD_grad.root
Select1.parquet
Select1_grad.parquet
Select2.parquet
Select2_grad.parquet
yields.json
yields_grad.json
result.json
result_grad.json


Backward
AOD.root
cfg.json
Select1.parquet
yields.json
yields_grad.json
result.json
result_grad.json

Forward

Backward

Forward
Backward
cfg.json
cfg_grad.json
AOD.root
AOD_grad.root
Select1.parquet
Select1_grad.parquet
Select2.parquet
Select2_grad.parquet
yields.json
yields_grad.json
result.json
result_grad.json
cfg.json
cfg_grad.json
AOD.root
AOD_grad.root
Select1.parquet
Select1_grad.parquet
Select2.parquet
Select2_grad.parquet
yields.json
yields_grad.json
result.json
result_grad.json

Forward
cfg.json
cfg_grad.json
AOD.root
AOD_grad.root
Select1.parquet
Select1_grad.parquet
Select2.parquet
Select2_grad.parquet
yields.json
yields_grad.json
result.json
result_grad.json
Backward
AOD.root
AOD_grad.root
cfg.json
cfg_grad.json
Select1.parquet
Select1_grad.parquet
Select2.parquet
Select2_grad.parquet
yields.json
yields_grad.json
result.json
result_grad.json

Forward

Backward

AOD.root

AOD_grad.root

cfg.json

cfg_grad.json

Autodiff system do this book-keeping for us
when data is in-memory

For async disk-based gradients, need to have efficient
infrastructure to book-keep (same for FAAS-type ideas)

Doesn't matter how gradient data is produced
(jax, TF, by hand, clad, ...)

this kind of infra is useful independentlty from how each
diffable sub-component are built

Select1.parquet

Select1_grad.par

Select2.parqu

Select2_grad.

yields.json

yields_grad.json

result.json

esult_grad.json

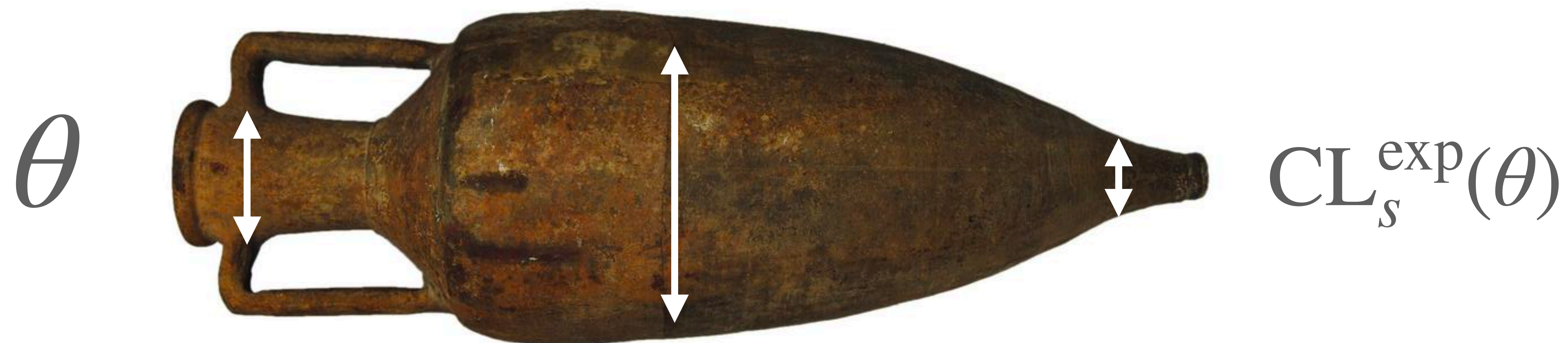
Mixing Forward and Backward

HEP analysis has *amphora* like shape:

- physics parameters and final objective: small O(10-100)
- intermediate representation large O(10^3-10^8)

$$\underbrace{\text{obj} = (\text{analysis}_\phi}_{\mathbb{R}^{10^3} \to \mathbb{R}^1} \circ \underbrace{\text{simulate}_\psi)}_{\mathbb{R}^{10} \to \mathbb{R}^{10^3}} (\theta)$$

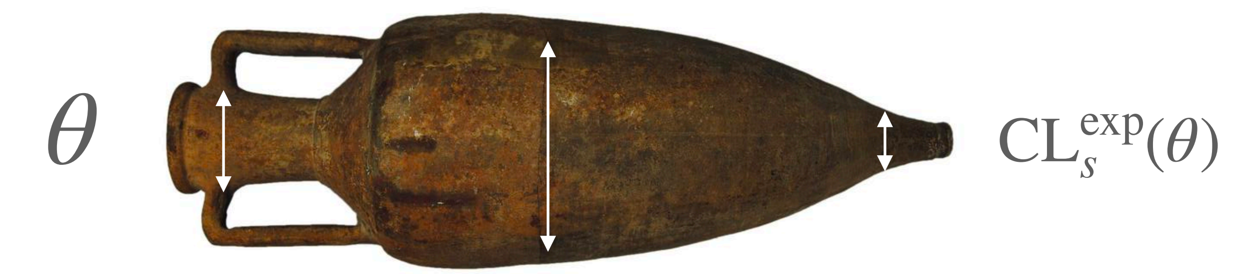
θ



$\mathrm{CL}_s^{\mathrm{exp}}(\theta)$

Mixing Forward and Backward

$$\mathrm{obj} = (\overset{\mathbb{R}^{10^3} \to \mathbb{R}^1}{\mathrm{analysis}_\phi} \circ \overset{\mathbb{R}^{10} \to \mathbb{R}^{10^3}}{\mathrm{simulate}_\psi})(\theta)$$



HEP analysis has *amphora* like shape:

- Jacobian $\partial \mathrm{obj}/\partial\theta$ is result of one wide and one tall Jacobian
- both can be efficiently computed <u>separately</u>

$\partial \mathrm{obj}/\partial\theta =$

J_{full}

=

obj vars

event vars

$\partial \mathrm{obj}/\partial \mathrm{event} = \dfrac{\partial(\mathrm{o}_1, \mathrm{o}_2, \ldots)}{\partial(e_1, e_2, \ldots)}$

J_{analysis}

pars

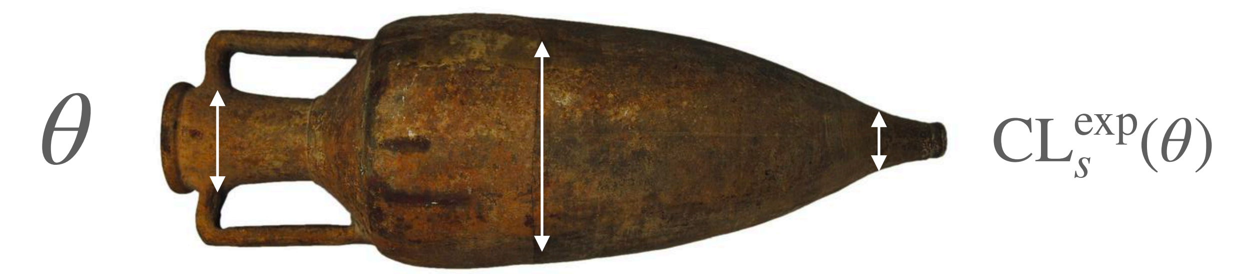
$\partial \mathrm{event}/\partial\theta =$

$\dfrac{\partial(\mathrm{e}_1, \mathrm{e}_2, \ldots)}{\partial(\theta_1, \theta_2, \ldots)}$

J_{simulate}

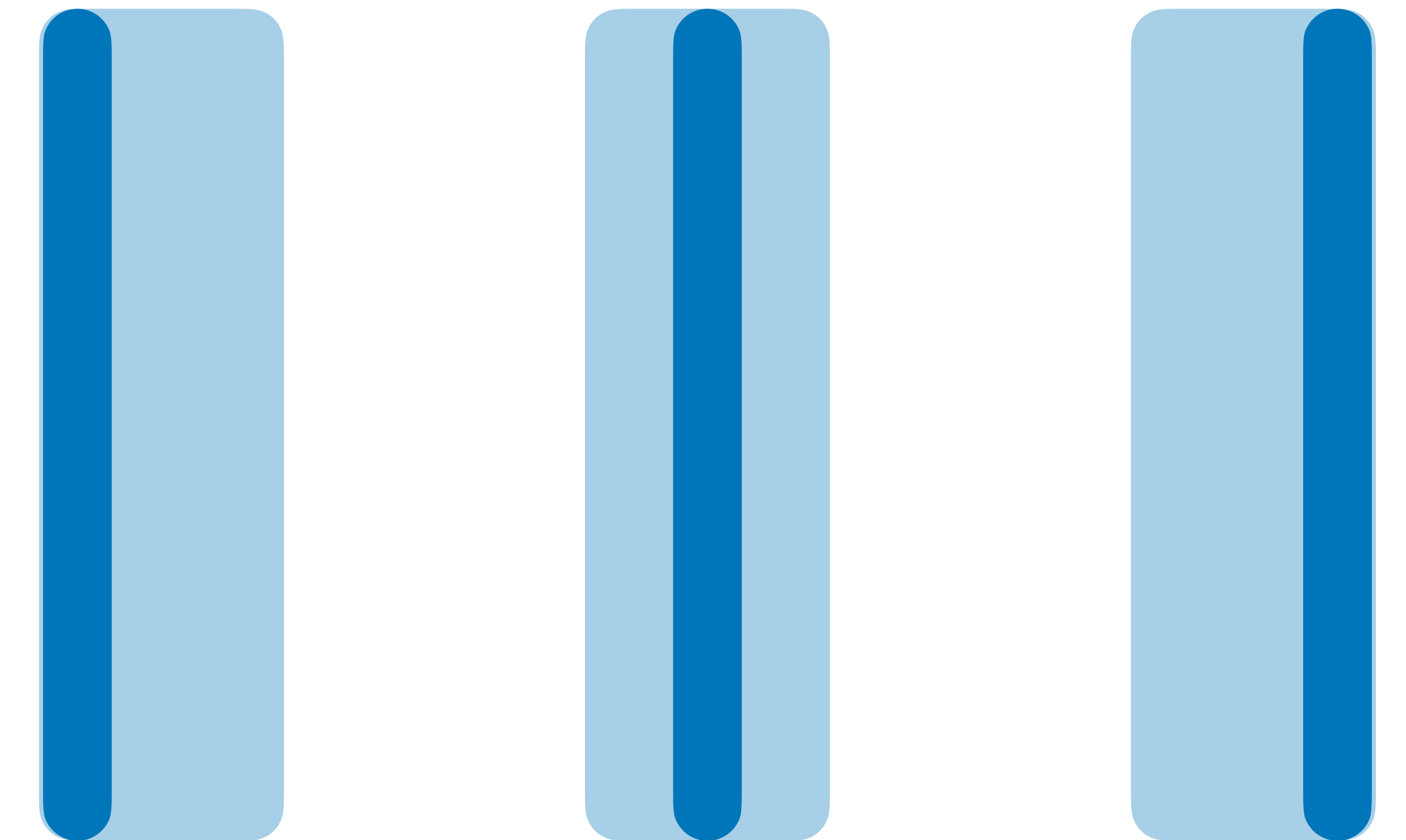
$\text{obj} = (\overset{\mathbb{R}^{10^3} \to \mathbb{R}^1}{\text{analysis}_\phi} \circ \overset{\mathbb{R}^{10} \to \mathbb{R}^{10^3}}{\text{simulate}_\psi})(\theta)$

Mixing Forward and Backward



tall Jacobian most efficiently computed w/ fwd-prop

- run differentiable simulator N times
  N = number of parameters / columns
- same θ, but different $\bar{\theta}$
- must only be recomputed if simulation configuration changes ψ

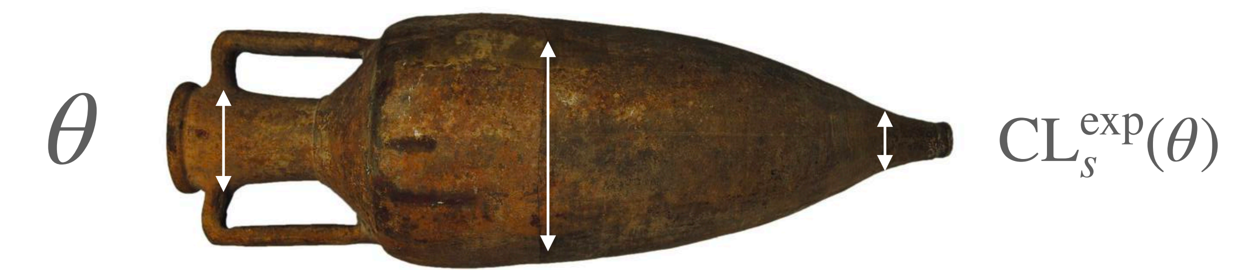


$$\text{event, column} = \text{simulate}(\theta, \bar{\theta})$$

$$\bar{\theta} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \quad \bar{\theta} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \quad \bar{\theta} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

$$\text{obj} = (\overset{\mathbb{R}^{10^3} \to \mathbb{R}^1}{\text{analysis}_\phi} \circ \overset{\mathbb{R}^{10} \to \mathbb{R}^{10^3}}{\text{simulate}_\psi})(\theta)$$

Mixing Forward and Backward

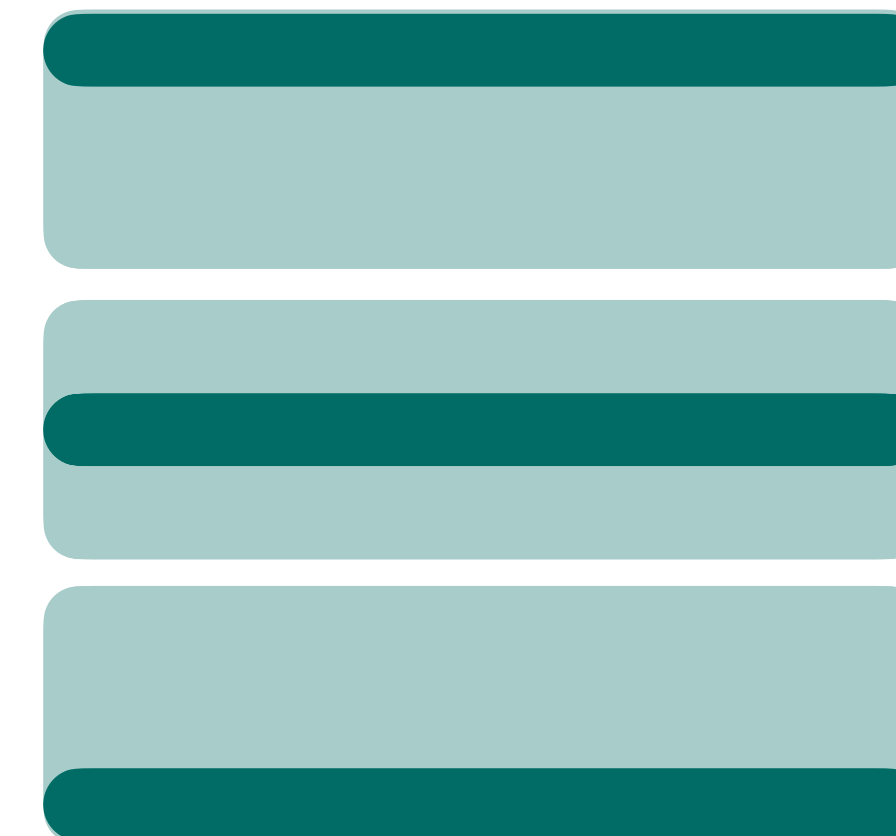


Wide Jacobian most efficiently computed w/ back-prop

- run forward, generate backfunc
- run backfunc N times, N = # of rows / objectives (often N=1)
- must only be recomputed if analysis configuration changes (ϕ)

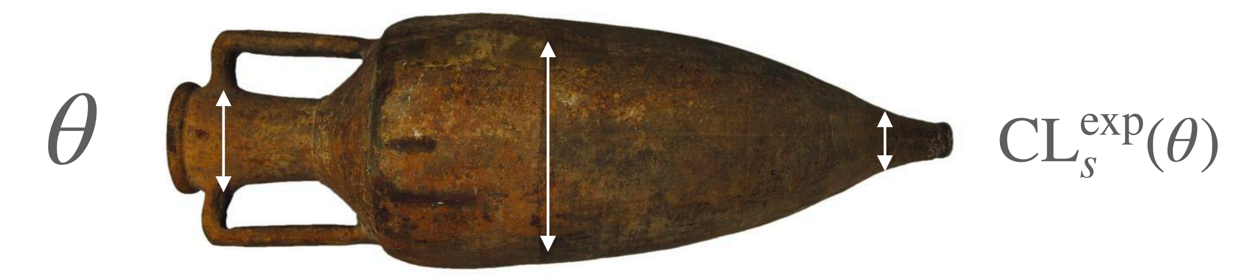
$$\text{obj}, \text{analysis}_{\text{back}} = \text{analysis}(\text{event})$$

$$\text{row} = \text{analysis}_{\text{back}}(\tilde{\text{obj}})$$



$$\mathrm{obj} = (\overset{\mathbb{R}^{10^3} \to \mathbb{R}^1}{\mathrm{analysis}_\phi} \circ \overset{\mathbb{R}^{10} \to \mathbb{R}^{10^3}}{\mathrm{simulate}_\psi})(\theta)$$

Mixing Forward and Backward



Total Jacobian then MxN (M=# objectives, N=# pars) inner products between rows and columns:

- recompute when either simulator or analysis change
- example of inner product in on structured event data

with clever use of fwd/bwd combo we can re-use a lot of information / limit which systems we touch

Building up a re-usable set of diffable buildig blocks

We have a few constucts we often use in analysis. Can we build a library that offers replacements? Which ones make sense?

slicing / "event views": attention (e.g. SpaNet)
fitting: diffing through optimization (e.g. neos)
histogramming: KDEs, softmax
ranking/sorting: convex polytops arxiv.org/2002.08871

```
counts, _ = np.histogram(data)
bestfit_pars = opt.mle.fit(loss,data,init)
slice_data = np[indices(data)]
```

```
counts, _ = gradhep.acc.histogram(data)
bestfit_pars = gradhep.opt.mle.fit(loss,data,init)
slice_data = gradhep.att(data, opts = ...)
```

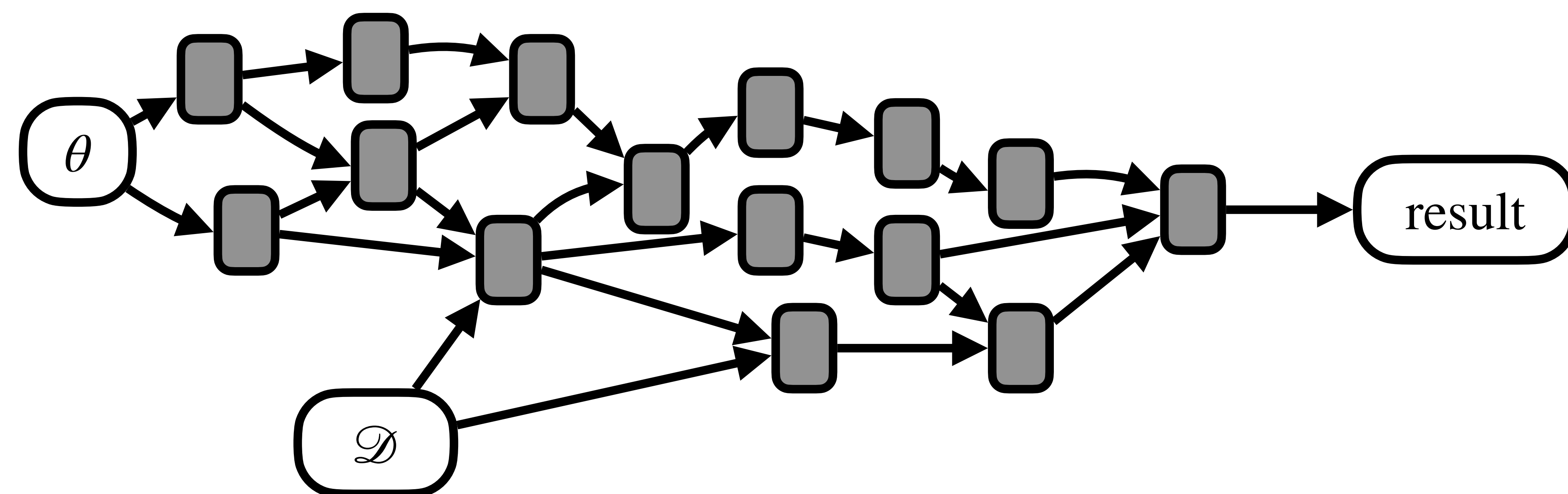
Open Question (to me)

Assume we have fully differentiable pipeline $o = f_\phi(\theta)$ with $\nabla_\phi f, \nabla_\theta f$

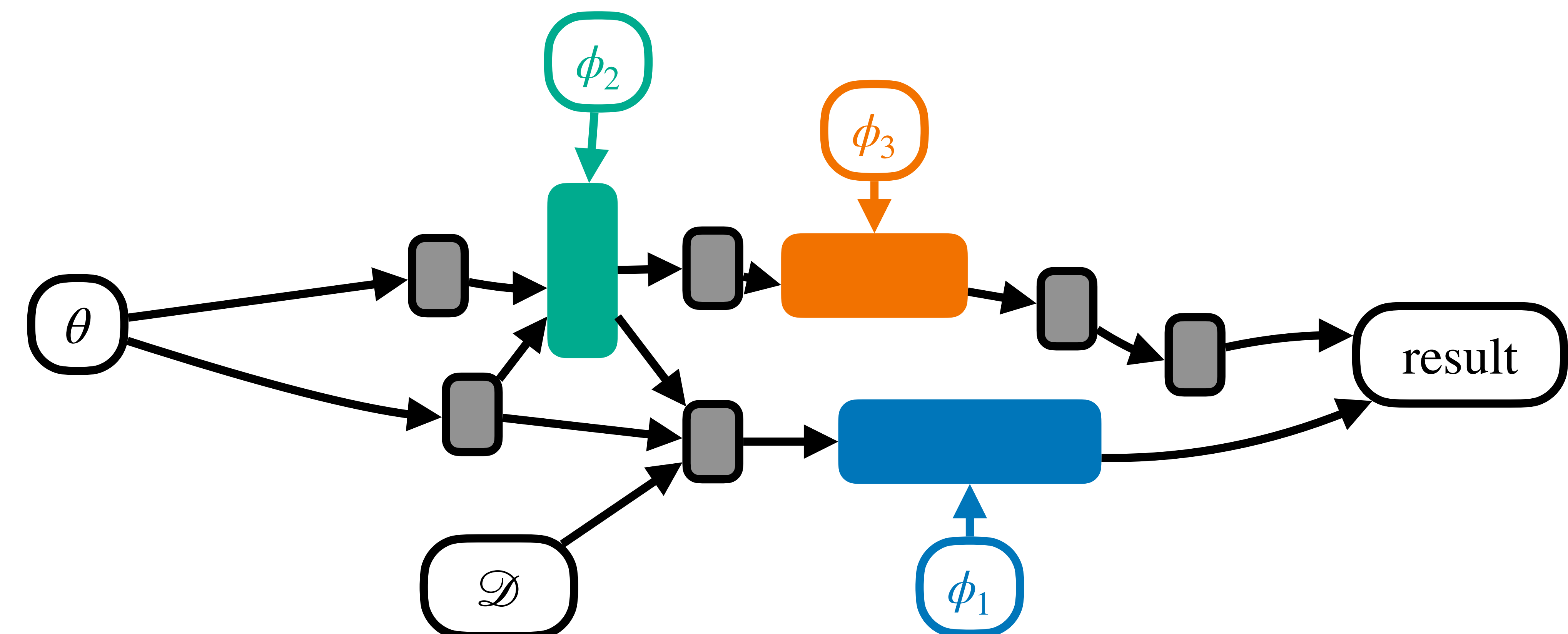
What does the loss-landscape look like to optimize e.g. ϕ

- does too much structure imply difficult landscape (local minima,..)
- part of magic of NN is severe overparametrization

end-to-end differentible
w/ fixed structure pipeline:



end-to-end differentible pipeline:
w/ oveparametrized "green-field" blocks



Dissemination

More people are becoming aware of possibility of differentiable analysis, seems not so crazy anymore

Maybe time for a white paper / "manifesto" / ... that gives a framework to build on?

ROOT(!) slide

HL-LHC Challenges

- More data doesn't just mean more analysis input data but also more stat power, means:
  - Can probe more complex models: more parameters / higher dimensions; sys uncertainty dominates: more studies / correlations / higher dimensions
  - Differential analysis / ML-optimization of parameters is expensive; end-to-end optimization even more
- Scaling of computing resources < scaling of data rate; growth of analysis needs >= growth of data rate?

7

Discuss

"differentiable" (versions of) `to_buffers` & `from_buffers`	"differentiable" learned embedders / flatteners `neurawk.flatten`
differentiable replacements (histograms, fits, sorting, ranking, selection)	on-disk autodiff / metadiff book-keeping of fwd/bwd
white paper?	loss-landscape question