Metadata thoughts
And some coffea tools

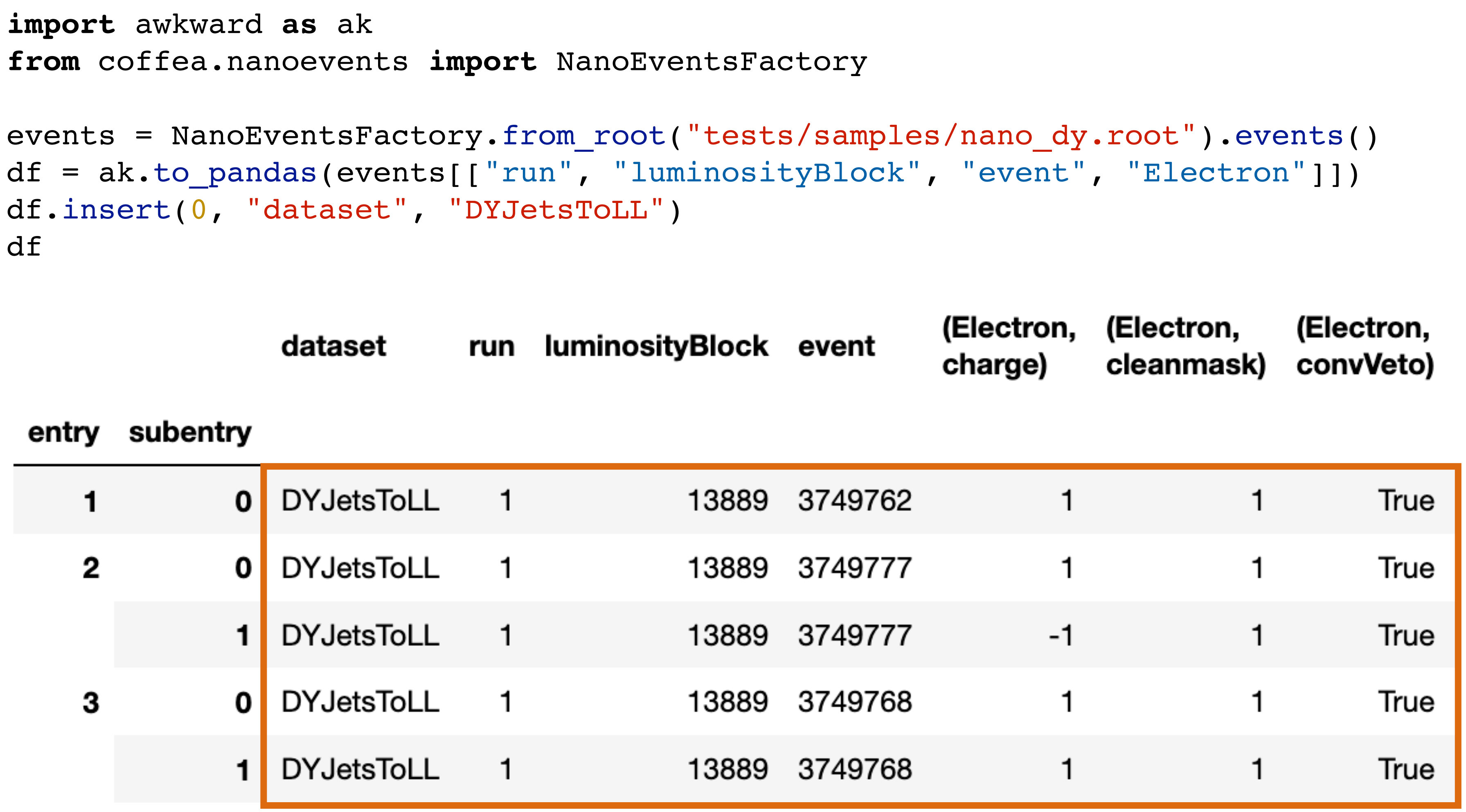
Nick Smith

HSF DAWG

17 February 2021

Introduction

- Data vs. metadata is really just an optimization detail



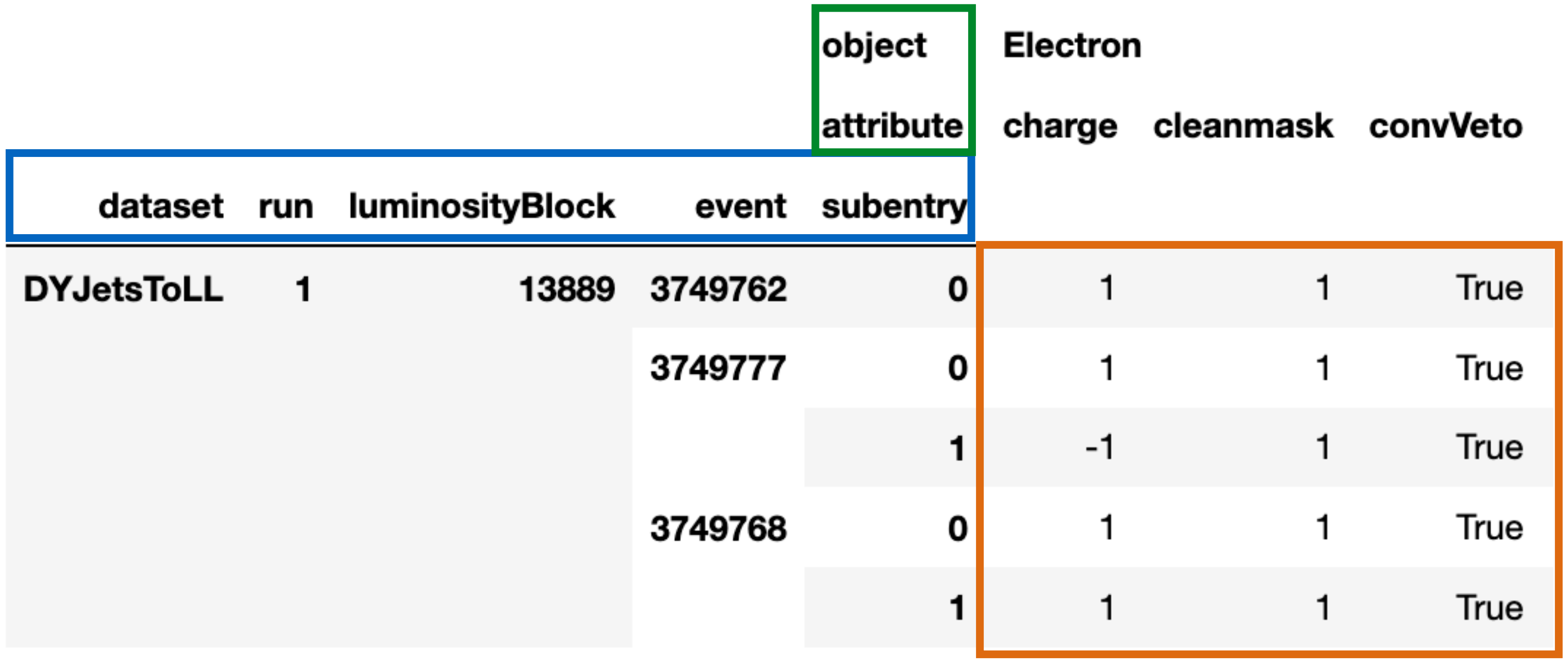
```python
import awkward as ak
from coffea.nanoevents import NanoEventsFactory

events = NanoEventsFactory.from_root("tests/samples/nano_dy.root").events()
df = ak.to_pandas(events[["run", "luminosityBlock", "event", "Electron"]])
df.insert(0, "dataset", "DYJetsToLL")
df
```

		dataset	run	luminosityBlock	event	(Electron, charge)	(Electron, cleanmask)	(Electron, convVeto)
entry	**subentry**							
1	**0**	DYJetsToLL	1	13889	3749762	1	1	True
2	**0**	DYJetsToLL	1	13889	3749777	1	1	True
	1	DYJetsToLL	1	13889	3749777	-1	1	True
3	**0**	DYJetsToLL	1	13889	3749768	1	1	True
	1	DYJetsToLL	1	13889	3749768	1	1	True

Introduction

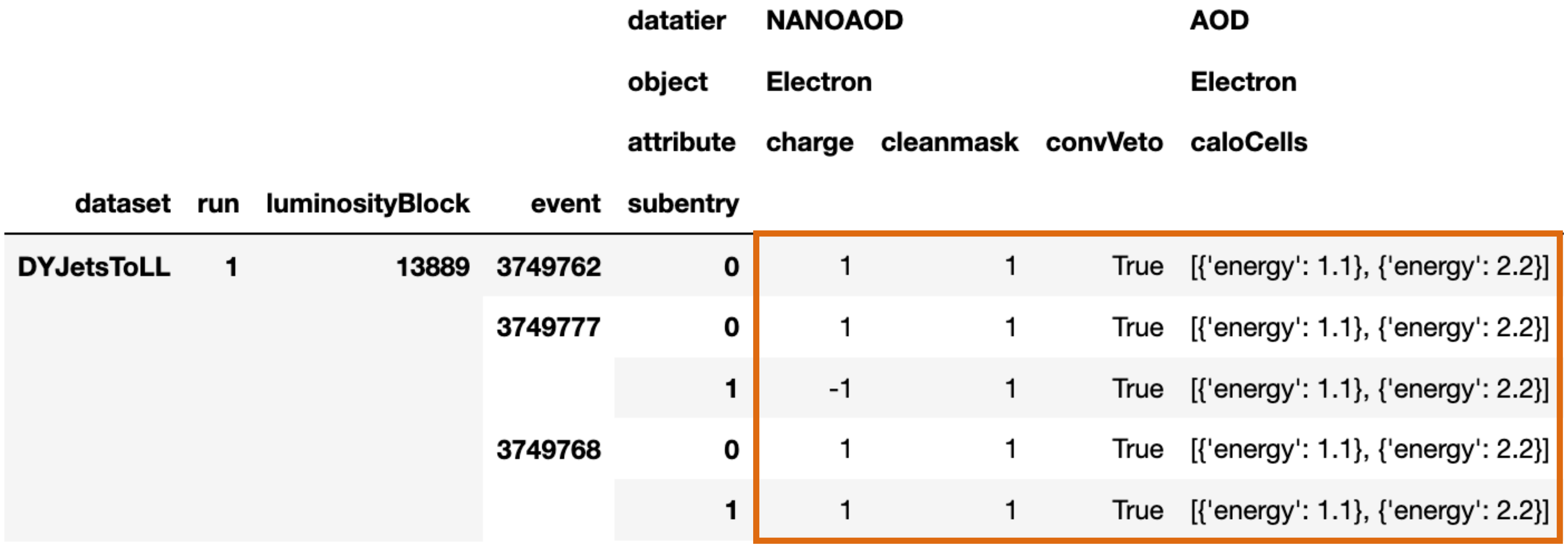
- Data vs. metadata is really just an optimization detail



				object	Electron		
				attribute	charge	cleanmask	convVeto
dataset	run	luminosityBlock	event	subentry			
DYJetsToLL	1	13889	3749762	0	1	1	True
			3749777	0	1	1	True
				1	-1	1	True
			3749768	0	1	1	True
				1	1	1	True

Introduction

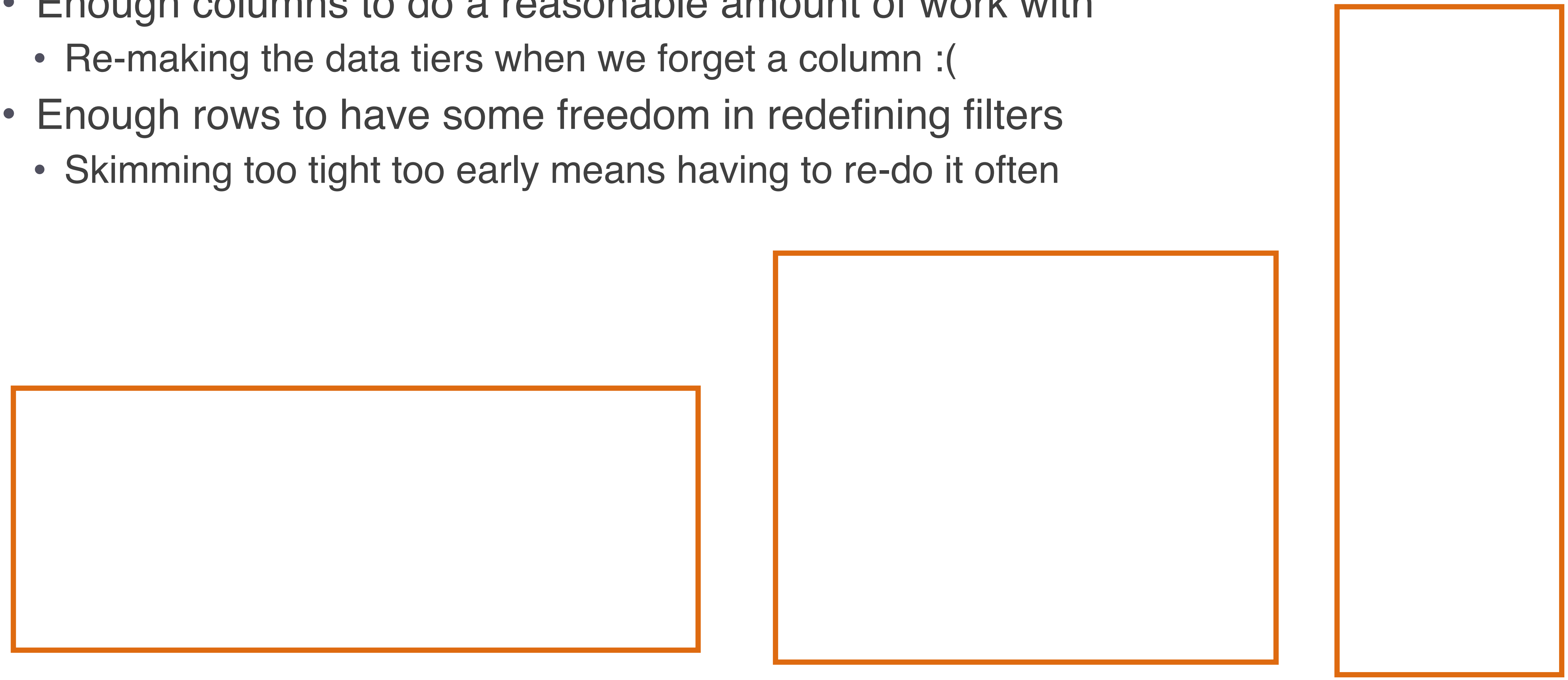
- Data tiers are column filters
- Files are row chunks
  - Splitting on an index level can sometimes speed up row filtering
- Sub-elements could go into:
  - Row indexer via explode (inverse of groupby)
  - Column indexer via pivot (unpleasant for irregular list sizes)



				datatier	NANOAOD			AOD
				object	Electron			Electron
				attribute	charge	cleanmask	convVeto	caloCells
dataset	run	luminosityBlock	event	subentry				
DYJetsToLL	1	13889	3749762	0	1	1	True	[{'energy': 1.1}, {'energy': 2.2}]
			3749777	0	1	1	True	[{'energy': 1.1}, {'energy': 2.2}]
				1	-1	1	True	[{'energy': 1.1}, {'energy': 2.2}]
			3749768	0	1	1	True	[{'energy': 1.1}, {'energy': 2.2}]
				1	1	1	True	[{'energy': 1.1}, {'energy': 2.2}]

Introduction

- We optimize data tiers and row chunks towards:
  - A target file size ~ area of rectangle
    - Why? Traditional filesystems can’t handle many small items? Object store to the rescue?
  - Enough columns to do a reasonable amount of work with
    - Re-making the data tiers when we forget a column :(
  - Enough rows to have some freedom in redefining filters
    - Skimming too tight too early means having to re-do it often

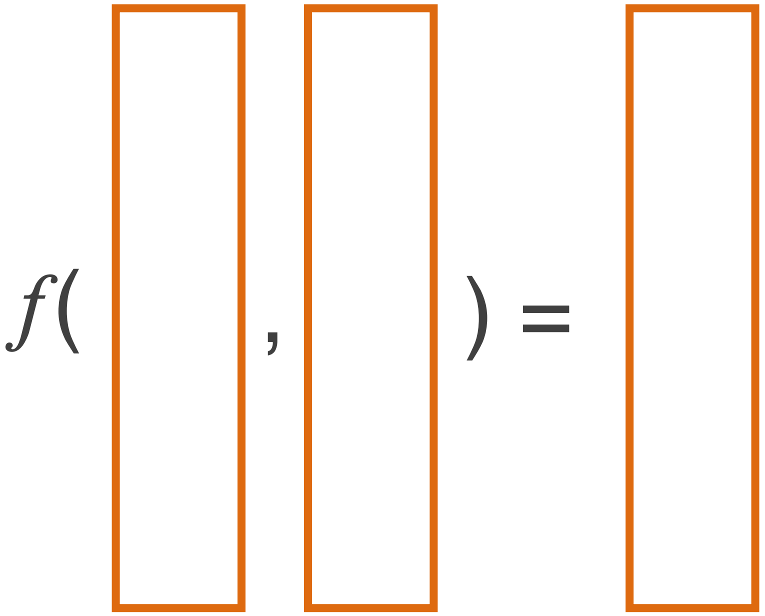


Introduction

- Can we join data tiers at analysis time?
  - Some columns may be wide because they contain many sub-elements
  - How do we analyze those alongside narrow columns?
  - Keep in mind even the row indexer metadata volume is huge

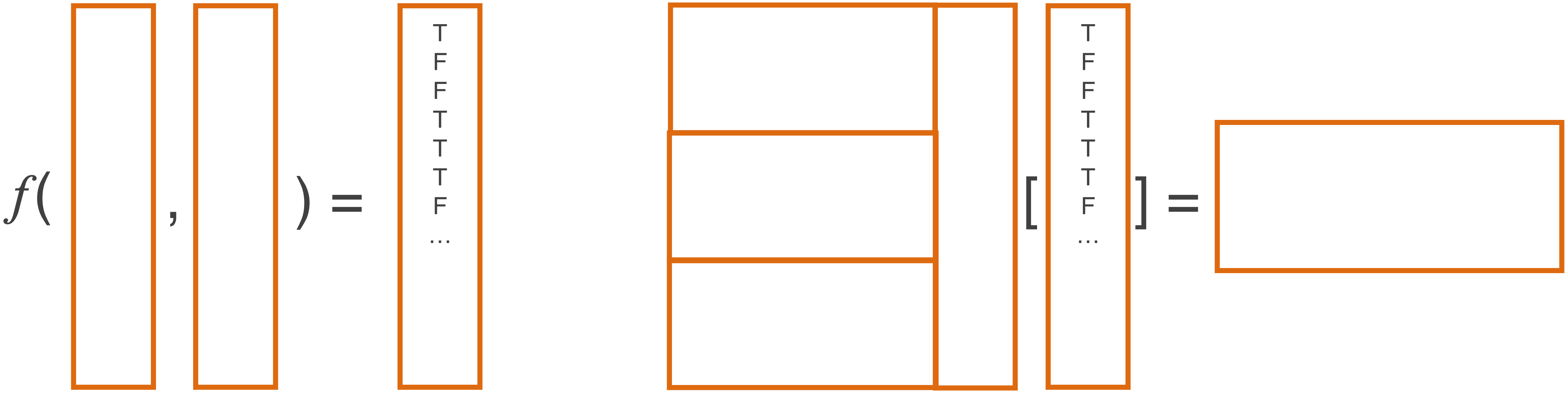
Introduction

- Non-event data (corrections) are interchangeable with columns
  - We can either use the function or its result
- The choice is again an optimization detail
  - Complex function, narrow output ➜ keep output
  - Simple function, wide output ➜ use function
  - Keep in mind decompression and bandwidth costs
    - Sometimes cheaper to recompute from values on hand



Introduction

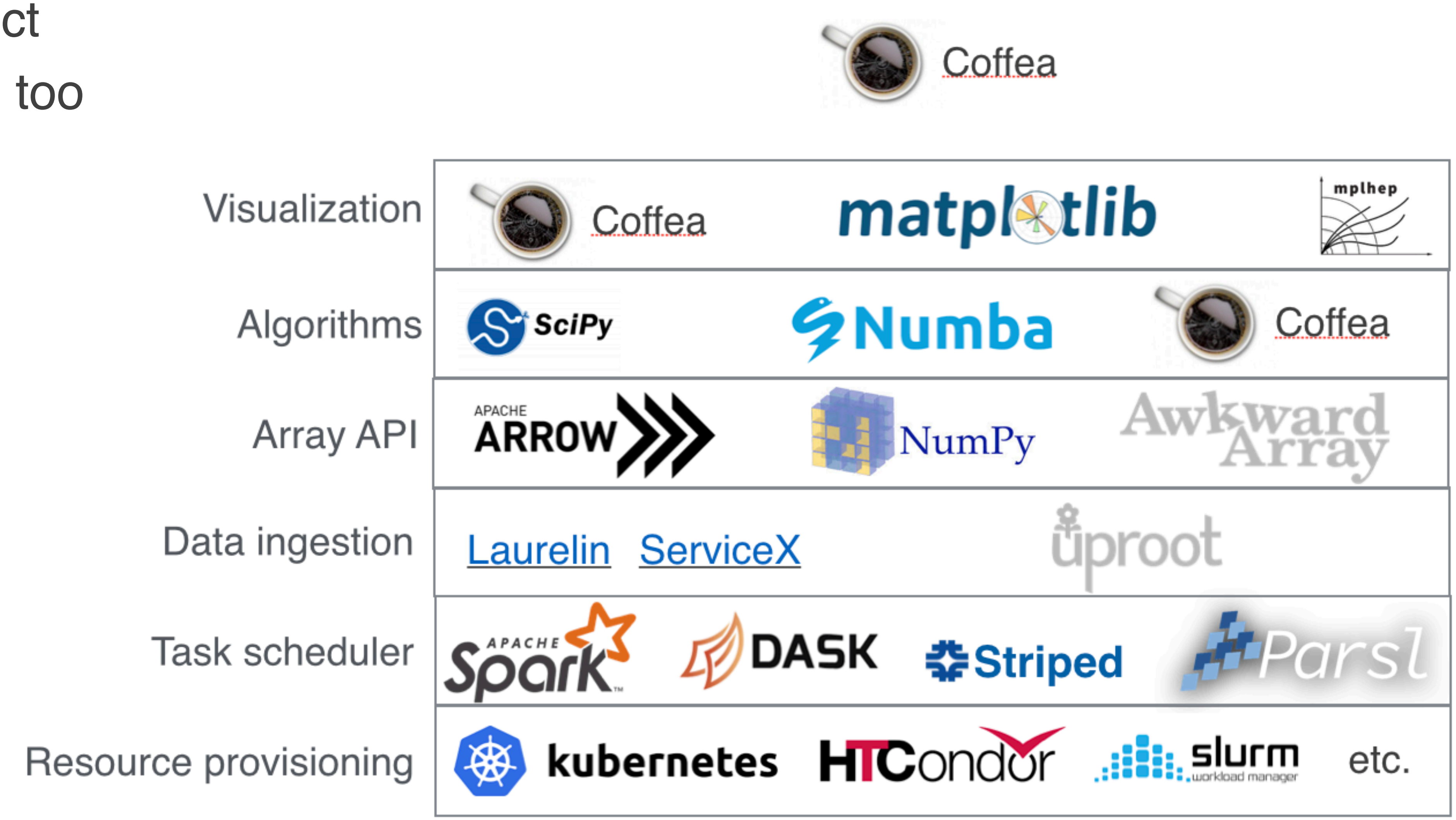
- Filters are interchangeable with boolean columns
- When do we want to save the function vs. column vs. filtered data tier?



Requisite advertisement

Coffea is:

- A package in the scientific python ecosystem
  - `$ pip install coffea`
- A user interface for columnar analysis
  - With missing pieces of the stack filled in
- A minimum viable product
  - We are data analyzers too
- A really strong glue



Coffea

Visualization: Coffea, matplotlib, mplhep

Algorithms: SciPy, Numba, Coffea

Array API: Apache Arrow, NumPy, Awkward Array

Data ingestion: Laurelin, ServiceX, uproot

Task scheduler: Apache Spark, DASK, Striped, Parsl

Resource provisioning: kubernetes, HTCondor, slurm workload manager, etc.

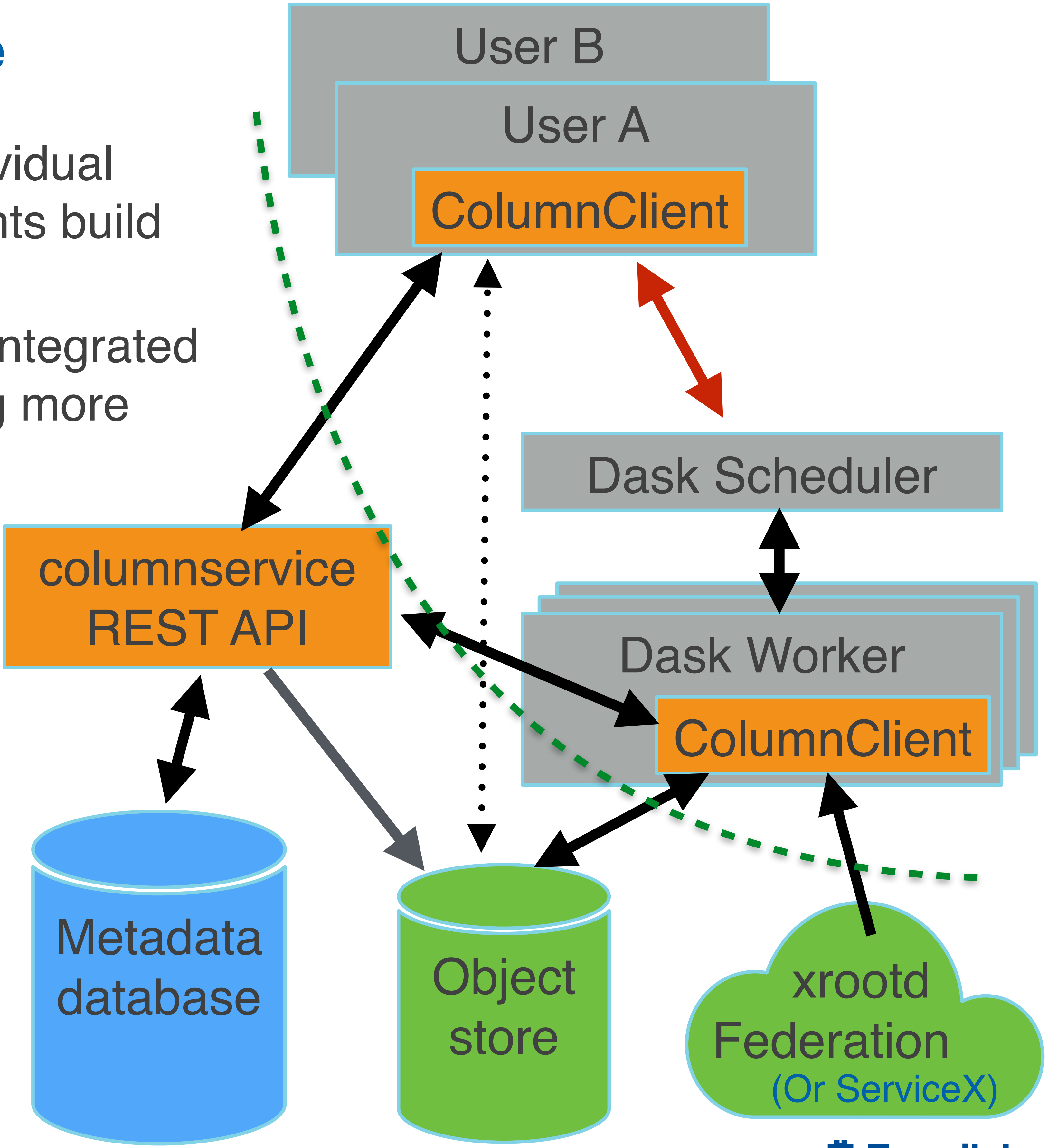
Coffea farm goals

- Data delivery is a main bottleneck for coffea at scale
- What could help:
  - Shared input cache at column granularity
  - Derived columns declared, only constructed and cached on access
    - Both projections (new columns) and filters (skims)
  - Unified metadata and dataset schema database
  - All declared and imported columns accessible *lazily*
  - Exportable columns

We want to design a scale-up mechanism for coffea users that removes the need to curate skims and re-run expensive algorithms over and over

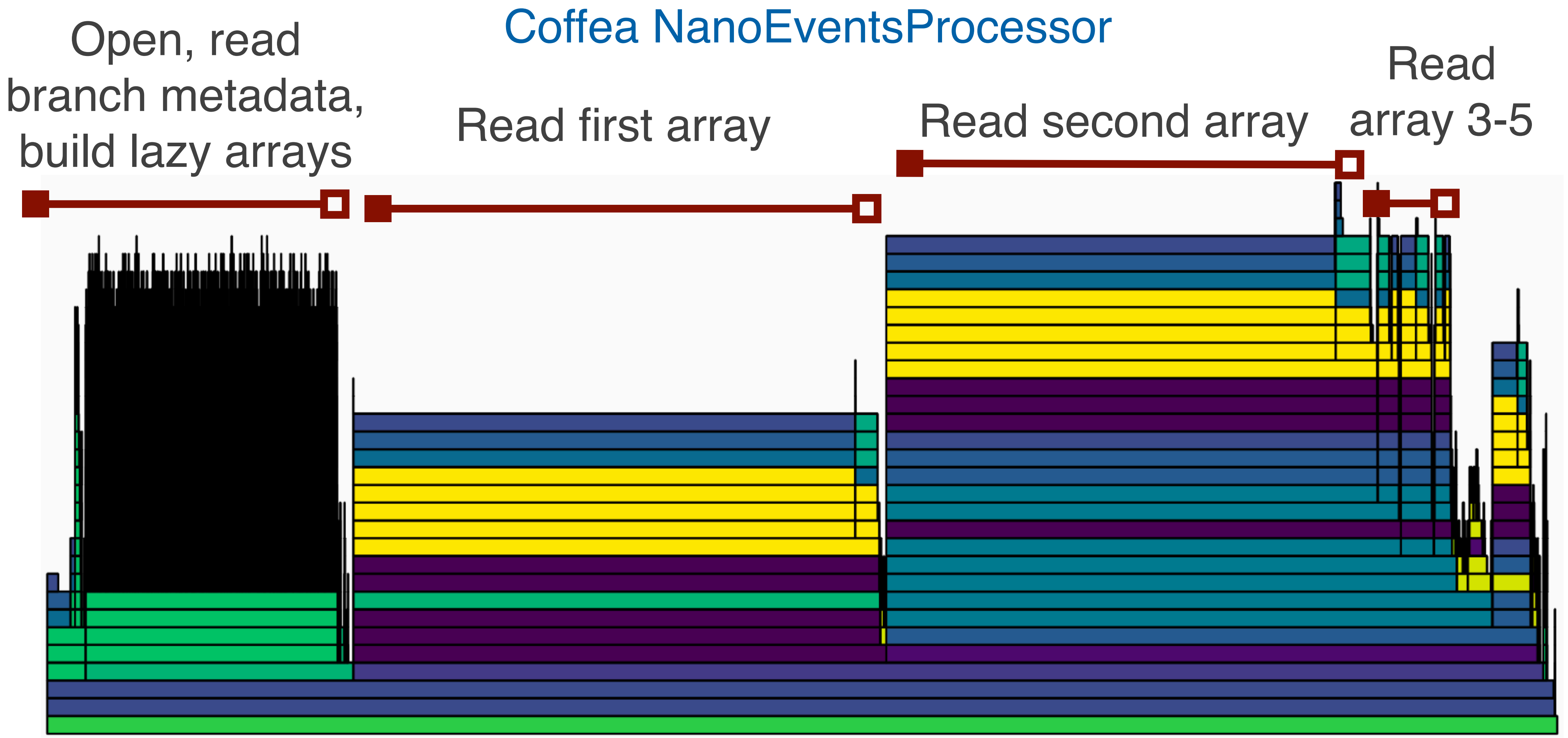
Columnservice prototype

- Manage the metadata of individual column objects and help clients build array chunks for processing
- Originally a k8s service with integrated dask cluster, now considering more lightweight solutions
  - Ideally ship columnservice with coffea, with e.g. SQLite for local and Postgres for site installs
  - User provides dask cluster, site provides object store (off the shelf)



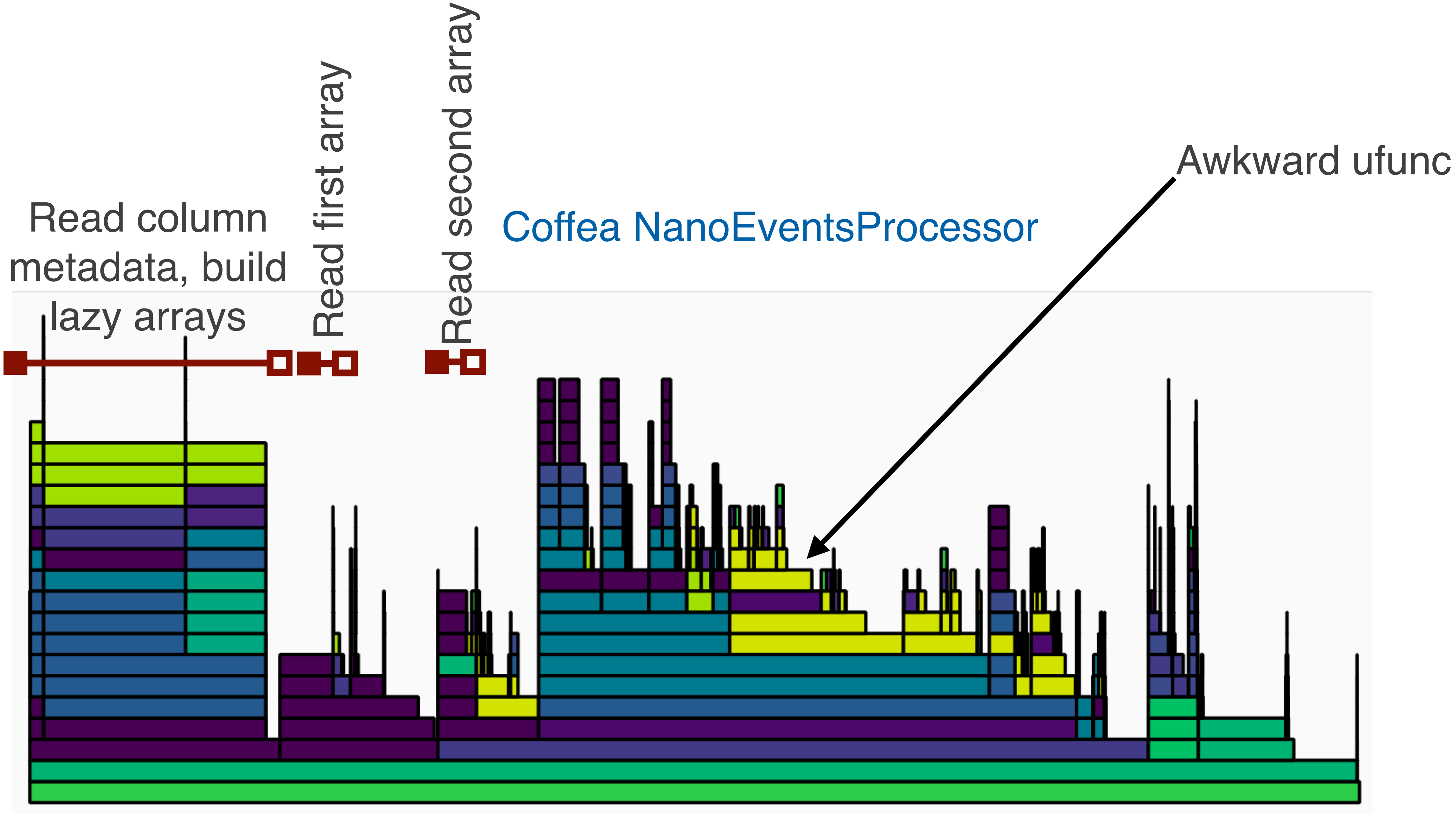
Columnservice case study: avoiding ingestion

- All inputs eventually come from ROOT files
  - True for the foreseeable future
- Reading and interpreting files with uproot is expensive
  - Even just opening and getting branch names can be significant
  - File byte-range caches take time to kick in, bad for small work packages



Columnservice case study: avoiding ingestion

With columnservice providing metadata, and an object store providing the array chunk, we start to see things other than read show up in the flame graph



Read column metadata, build lazy arrays

Read first array

Read second array

Coffea NanoEventsProcessor

Awkward ufunc

Persisting non-event data

- We want a service that can decide when to cache function output
- Necessary ingredient: persist-able function definitions
  - Bonus: analysis preservation?
- Coffea distributed executors all use cloudpickle
  - No forward or backward compatibility guarantees for pickled python functions
  - Good for getting user code to scale-out mechanisms, bad for persistence
- Correctionlib may be a possible solution
  - Store corrections in JSON format with a flexible schema
  - Implement evaluator(s)
    - High-performance scalar function evaluator provided by library
    - High-level types handled by extension libraries
  - Join the fun: https://github.com/nsmith-/correctionlib

```
def f(*args: Union[str,int,float]) -> float:
    return ...

double Correction::evaluate(const std::vector<std::variant<int, double, std::string>>& values) const;
```

Summary

- Its useful to think abstractly in terms of data frames
- Many analysis workflow decisions are optimization problems
  - It is not easy in many frameworks to adjust the approach
  - It would be nice if optimization choices were made automatically
- Coffea continues to investigate novel approaches to these issues