Columnar data analysis in HEP

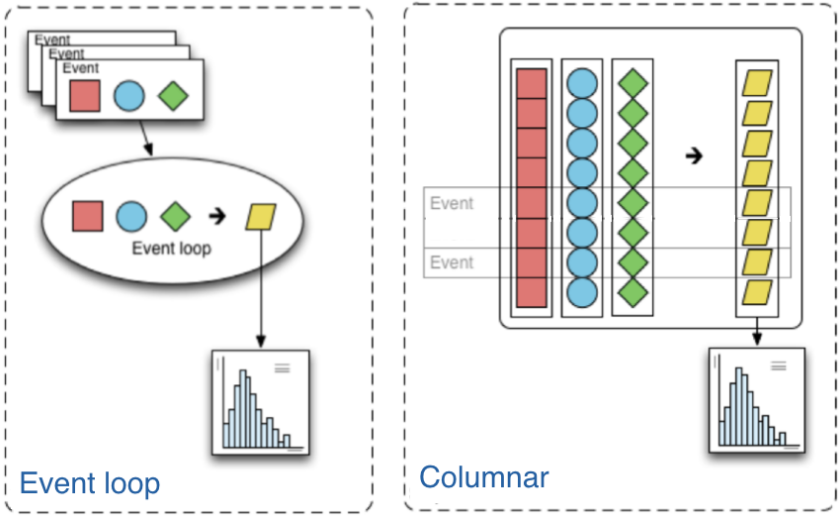
challenges and prospects

Nikolai Hartmann

LMU Munich

February 15, PUNCHLunch

Columnar data analysis / Array programming



[1] Plot from https://coffeateam.github.io/coffea/concepts.html

Array Programming History

```
life ← {⊃1 ⍵ ∨.∧ 3 4 = +/ +⌿ ¯1 0 1 ∘.⊖ ¯1 0 1 ⌽¨ ⊂⍵}
```

- Writing program logic *array/column-at-a-time* instead of *row/event-at-a-time*
- Concept exists since long before i was born (e.g. APL in the 60/70s)
  → very niche, main use nowadays probably in coding challenges
- Fortran also supports array programming, but not the main paradigm used (?)
  → https://fortran-lang.org/cs/learn/rosetta_stone
- Became mainstream probably because of
  - **Interactive** analysis in interpreted languages (Matlab, R, **Python with numpy/pandas**)
  - The rise of **Machine learning** since the 2010s (now via tensorflow, torch, jax)
- People like it!
  - New students are increasingly familiar with these concepts
    ... even before i learned about this - 90% of my PhD thesis analysis used `TTree::Draw`
    → basically also amounts to thinking *array-at-a-time*
  - Last step of analysis already widely done with numpy and pandas
    → e.g. many analyzers at Belle II more familiar with pandas than ROOT

But Why?

Advantages

- Predefined operations, no for loops!
  → Move slow bookkeeping out of the event loop
  → Write analysis code in python instead of C++
- Run on contiguous blocks in memory
  → fast (good for CPU cache, vectorization possible)
- Advances in tools in recent years
  → data science/machine learning
  → also in HEP: uproot, awkward array, coffea

Disadvantages

- Arrays need to be loaded into memory
  → need to process chunk-wise if amount of data too large
- Some operations complex to implement
  (e.g combinatorics, nested selections, variable length lists per event)

But for Particle Physics we want

- Objects
  → don't want to manually operate on px, py, pz, ...
- Variable length lists
  → each event has a list of Electrons, Muons, Jets, ...
- Cross references
  → `Electrons.trackParticles.pt` should give me the right thing
  ... even if this is stored in a different column/array
- The solution: move from array of structures to **structures of arrays**
- Side note: translating this to storage also facilitates **I**nteroperable and **R**eusable data
  → structure becomes metadata - can be described by simple standard like json
  → bulk data are just plain arrays - can be stored in a variety of standard formats

Awkward Array

https://awkward-array.org

- Developed by Jim Pivarski and others (most funding from IRIS-HEP)
- Supports nested records (`RecordArray`)

  e.g. `Events -> [Electrons -> pt, eta, phi, ..., Jets -> pt, eta, phi ...]`
- Variable length lists (`ListOffsetArray`)
- Cross references via indices (`IndexedArray`)
- Missing values via indices or masks (`IndexedOptionArray` , `Byte/BitMaskedArray`)
- Behavior/Dynamic quantities
  e.g. Lorentz vectors - can add vectors, calculate invariant masses, ...
  → provided by the `vector` package
- Everything operates on pure arrays of numbers
  → *structure-of-arrays* instead of *array-of-structs*
- ROOT files via `uproot` , conversion to/from `ROOT.RDataFrame` and `cppyy` possible

Escape hatch - just-in-time (JIT) compilation

- Sometimes operations may be difficult to wrap your head around (e.g. combinatorics)
  - Some awkward array functions exist (e.g. `ak.cartesian`, `ak.combinations`)
  - But may prefer to use loops (also for performance reasons, e.g. skipping combinations)
- JIT compilers can help, e.g. Numba
  - Supports numpy
  - Supports awkward array
    - passing awkward arrays "just works"
    - returning awkward arrays either via `ArrayBuilder` or flat arrays + `ak.unflatten`
- For existing c++ may want to manually build python wrapper, e.g. via
  - pybind11
  - cppyy
  - ROOT

From columnar analysis to declarative analysis

- Columnar data analysis gets quickly transformed **declarative** analysis
  → removing loops, we are left with cut definitions and (high level) transformations
- Important tools
  - ROOT RDataFrame
  - dask-awkward
    → don't execute operations eagerly, but build computation graph
- The dream: Completely decouple execution from the declared analysis
  → can apply optimzations (e.g. fuse operations, jit compile)
  → execute single/multithreaded, on single machine or cluster
  → maybe even move to a more database-like system (e.g. serviceX)
- I believe we are not quite there yet and a bit unclear if this is the future
  → executing the same code on many files in parallel is always an option

Dask

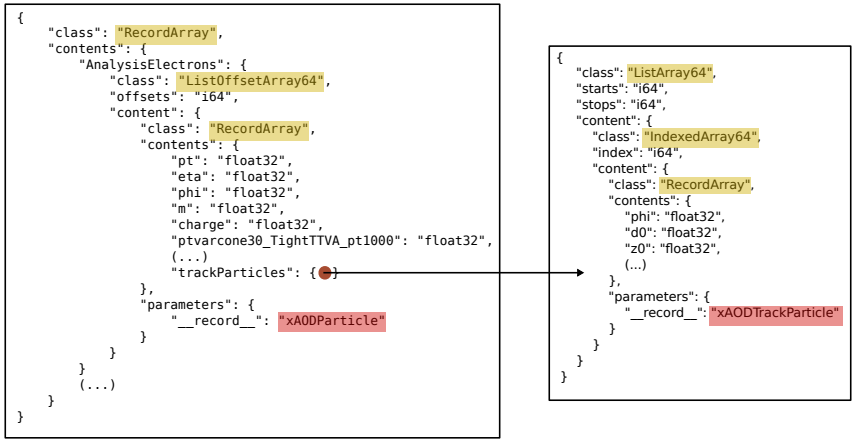
https://www.dask.org

- Python framework for **parallel** computing
- **Low-level** interface via **delayed** and **futures**
  → define custom computation graphs in python
- **High-level**: **distributed** equivalents of **numpy arrays** and **pandas DataFrames**
  → distributed **awkward array** in development
- Live **dashboard** with computations/status/profiler
  → very useful for debugging and optimizing (also looks nice)
- `dask-jobqueue` to spawn dask cluster on top of a **batch system**
  → slurm, HTCondor and more supported
  → single jobs, start immediately with as many workers as you got
- **ROOT RDataFrame** can also use dask as a backend
  → dask as a universal interface to interactive parallelism?

Some experiences with ATLAS `DAOD_PHYSLITE`

- `DAOD_PHYSLITE`: reduced ATLAS data format with (currently) 10kb per event
  → standard calibrations applied
  → readable (with caveats) without of ATLAS software stack
  → could be used to analyse with python tools and columnar data analysis
- At CMS there is some success with a similar `NanoAOD` format (2kb per event)
- The `coffea` framework provides many functionalities
  → `coffea.nanoevents` for representing such formats as awkward array (including cross references, lazy loading etc)
  → developed prototype schema to support `DAOD_PHYSLITE` with this

Represent the PHYSLITE event-data-model as an awkward array



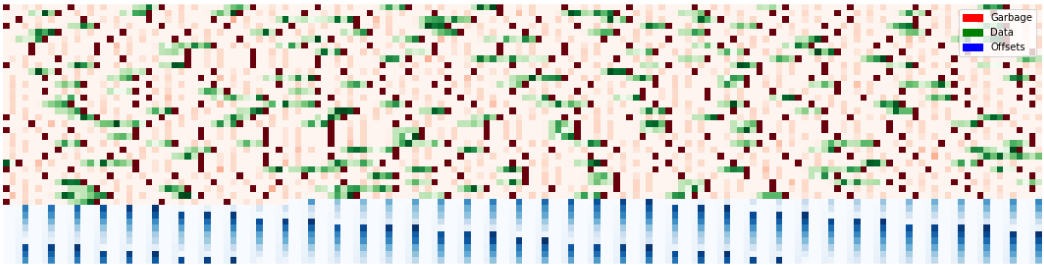

```
>>> # pt of the first track particle of each electron in events with at least one electron
>>> Events[ak.num(Events.AnalysisElectrons) >= 1].AnalysisElectrons.trackParticles.pt[:,:,0]
<Array [[2e+04], [2.13e+04, ... [1.73e+04]] type='225 * var * float32'>
```

reading ROOT TTree data with uproot

- uproot can now read basically everything we need for `DAOD_PHYSLITE`
- fundamental types and 1D arrays/vectors are fine
  → can with a few tricks read them as a whole block
- `vector<vector<...>>` requires loop
  - now working reasonably efficient using `awkward forth`
  - ... but often buggy, e.g. uproot#951 (seen in p5631 PHYSLITE files)

Intermezzo: why ROOT TTree is not ideal

Binary basket[1] data for electron pt (`vector<float>`):

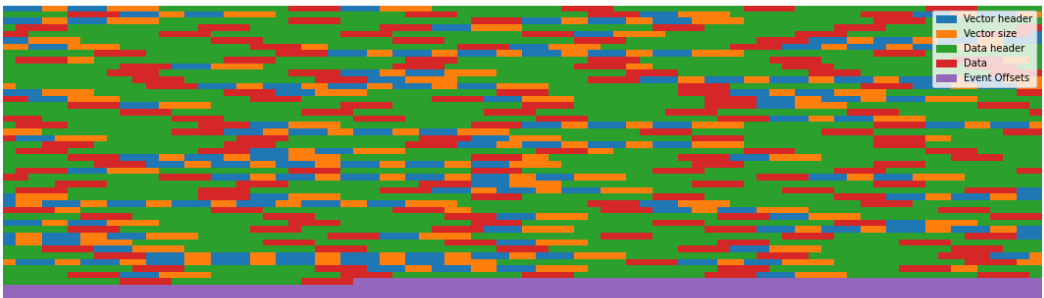


"Garbage": Header (telling us "this is a vector") and number of bytes following (redundant)
→ **green and blue marked data is the only information we actually need**

[1] block of compressed data in ROOT TTree, typically containing data for multiple events

even worse for higher dimensional vectors and objects

Binary basket data for electron-track cross references (`vector<vector<ElementLink>>`)



→ **red and orange marked data is the only information we actually need**

What's better?

Loading times for all columns ($\approx$ 1000) of 10k DAOD PHYSLITE events

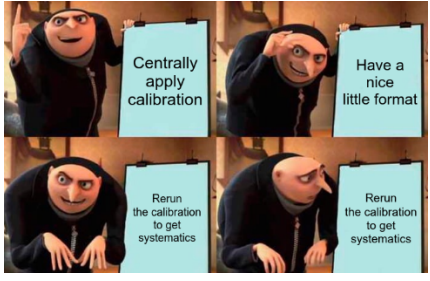
Format	Compression	Dedup. offsets	Size on disk	Execution time
(Up)root	zlib	No	117 MB	$6.0\,\mathrm{s}$
(Up)root (large baskets)	zlib	No	116 MB	$5.0\,\mathrm{s}$
Parquet	snappy	No	121 MB	$0.6\,\mathrm{s}$
Parquet	snappy	Yes	118 MB	$0.6\,\mathrm{s}$
HDF5	gzip	No	101 MB	$2.0\,\mathrm{s}$
HDF5	gzip	Yes	89 MB	$1.6\,\mathrm{s}$
HDF5	lzf	No	137 MB	$1.5\,\mathrm{s}$
HDF5	lzf	Yes	113 MB	$1.1\,\mathrm{s}$
npz	zip	No	92 MB	$2.0\,\mathrm{s}$
npz	zip	Yes	82 MB	$1.5\,\mathrm{s}$

Parquet seems especially promising, but all tested formats faster than Up(root)
(Note: constant overhead for Uproot, will be less significant for larger number of events)

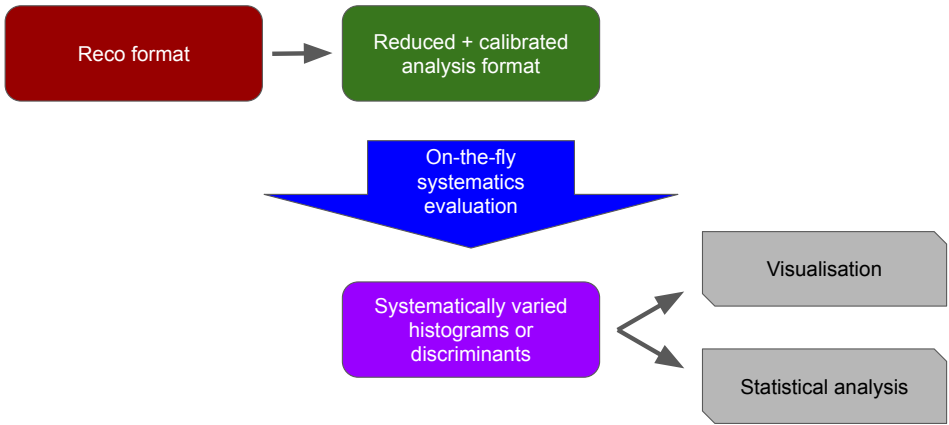
DAOD PHYSLITE Prototypes for **ROOT RNtuple** exist and we expect comparable performance to Parquet, stay tuned!

Challenge - Systematics

- Vision: evaluate systematic variations on the fly on PHYSLITE
  $\rightarrow$ avoid to store $N_{\text{systematics}}$ copies
- Problem: currently run during calibration (already done in PHYSLITE)
  $\rightarrow$ need to find a way to parametrize based on "nominal" calibration
  $\rightarrow$ ideally not dependent on too many variables
  $\rightarrow$ could also reduce number of needed columns



Systematics: The Vision



[1] from Teng Jian Khoo's summary at the Analysis Ecosystems Workshop

Possible solution: wrap existing C++ code

```
tool = ElectronEfficiencyCorrectionTool()
tool.initialize()

# pass in awkward array of electrons
# get back awkward arrays of scale factors
sf, sf_total, status = tool.compute(events.Electrons)
```

- ATLAS has a streamlined framework for systematic corrections
- Want to avoid rewriting all that code
- But current code is too slow for on-the-fly systematics in interactive analysis
- Plan: wrap the existing C++ code into a columnar interface
  → Nils' presentation at CHEP23
  → upcoming Poster by Matthias Vigl at ACAT24

Summary

- Columnar analysis/Array programming greatly benefits
  - Ease of use - write code in python instead of C++
  - Interactive exploration, thus increasing developer productivity
  - Potentially faster code (no bookkeeping in the event loop, CPU cache, simd, ...)
  - The **I** and **R** in FAIR through interoperable and reusable tools and storage formats
- Tools for HEP specific needs
  - Awkward Array for numpy-like analysis with more structured data
  - uproot for reading ROOT files
  - RDataFrame and dask-awkward for declarative, parallelizable analysis
- ATLAS PHYSLITE is a great case study
  - How to deal with more complex objects?
  - How to combine columnar analysis with legacy C++ code?