Enrico Guiraud
ROOT team, EP-SFT, CERN

October 27, 2022
ACAT 2022

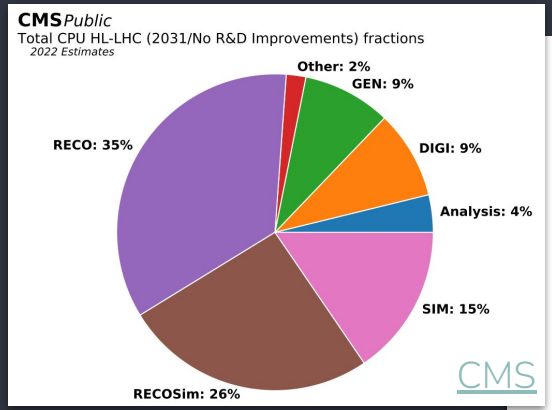
Simpler, faster and bigger

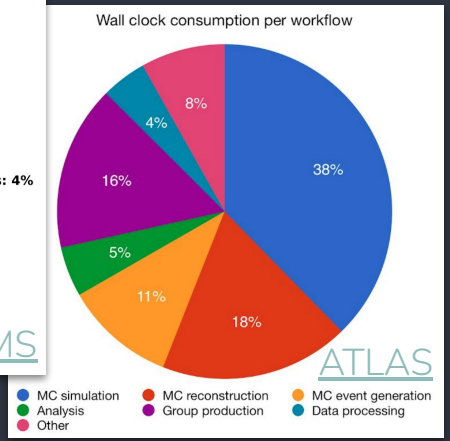
HEP analysis in the LHC Run 3 era

Motivation, context

Why we care

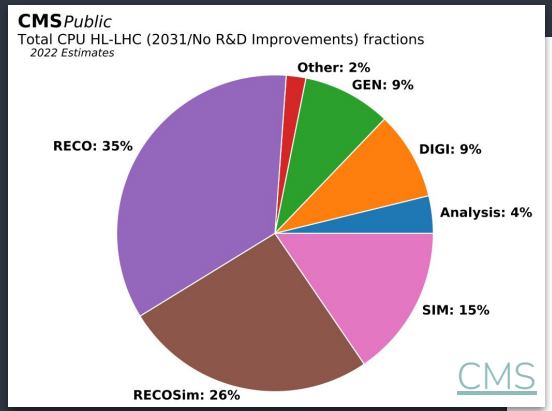
CPU time

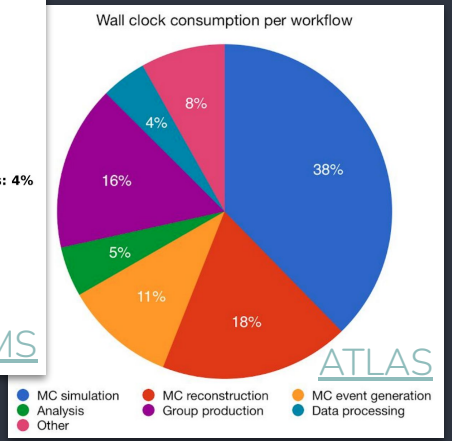




Why we care

CPU time

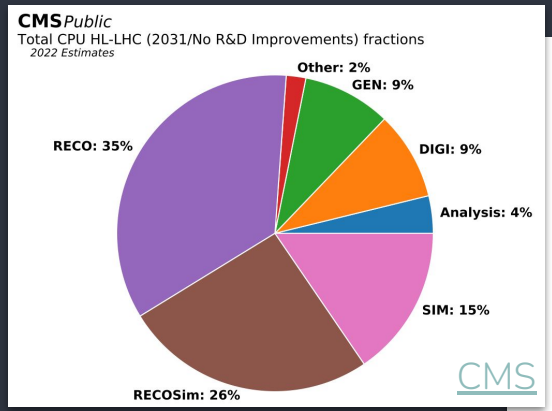


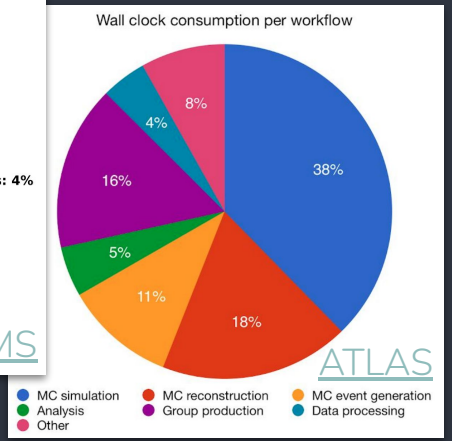


PhD student time

Why we care

CPU time





PhD student time

- All analysis software becomes 2x faster -> 2x jobs on the same resources
- One analysis becomes 10x (or 1000x) faster -> new explorations possible

Conversely: if your analysis had to process 10x data today, would you be ok?

The Triforce of analysis frameworks

- **Ergonomics**: onboarding, docs, debugging (correctness and bottlenecks), extensibility, prototyping, making simple things simple, difficult things possible.
- **Performance**: best possible throughput and hardware utilization out of the box.
- **Sustainability**: validation, stability, user support, bug fixes – over years (decades?).

The Triforce of analysis frameworks

- **Ergonomics**: onboarding, docs, debugging (correctness and bottlenecks), extensibility, prototyping, making simple things simple, difficult things possible.
- **Performance**: best possible throughput and hardware utilization out of the box.
- **Sustainability**: validation, stability, user support, bug fixes – over years (decades?).

Choice of programming language(s) is important for all three aspects.

The Triforce of analysis frameworks

- **Ergonomics**: onboarding, docs, debugging (correctness and bottlenecks), extensibility, prototyping, making simple things simple, difficult things possible.
- **Performance**: best possible throughput and hardware utilization out of the box.
- **Sustainability**: validation, stability, user support, bug fixes – over years (decades?).

Choice of programming language(s) is important for all three aspects.

End users will mostly care about ergonomics (see also N. Smith),
service managers and experiment coordination value performance and sustainability.

The Triforce of analysis frameworks

- **Ergonomics**: onboarding, docs, debugging (correctness and bottlenecks), extensibility, prototyping, making simple things simple, difficult things possible.
- **Performance**: best possible throughput and hardware utilization out of the box.
- **Sustainability**: validation, stability, user support, bug fixes – over years (decades?).

Choice of programming language(s) is important for all three aspects.

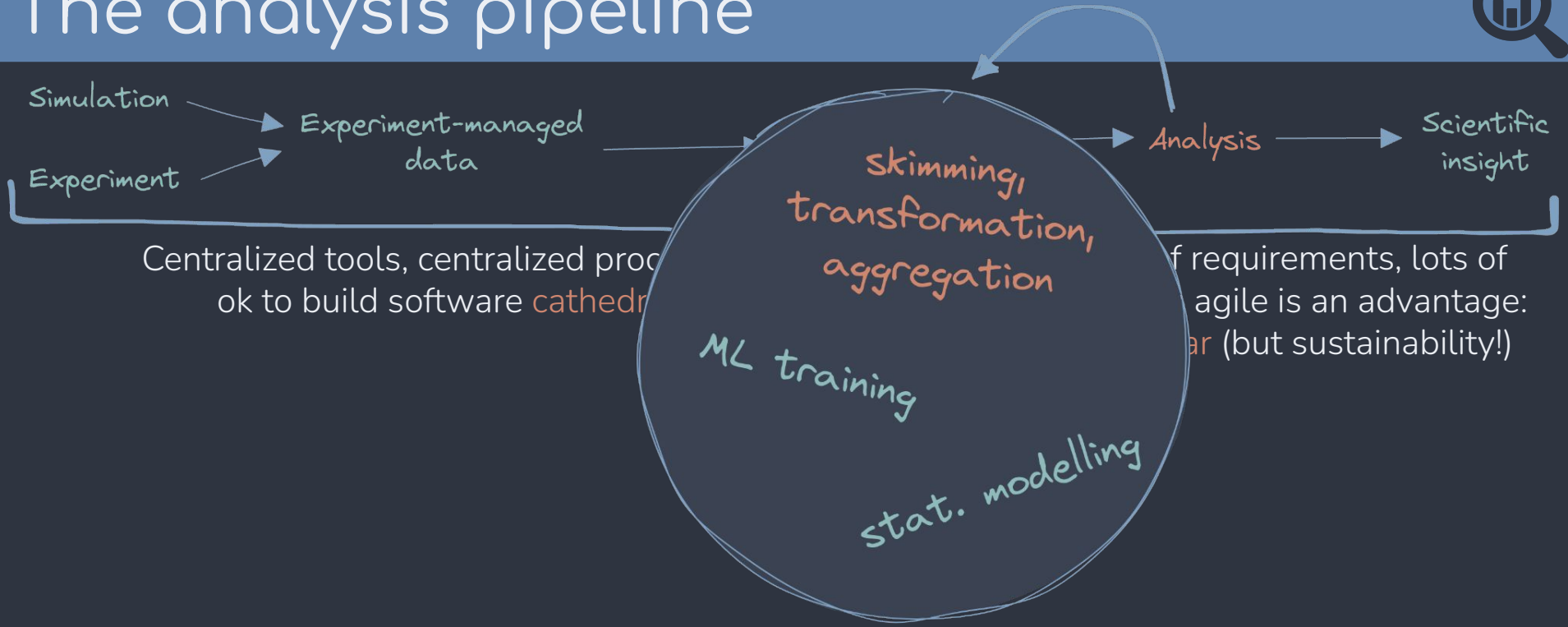
End users will mostly care about ergonomics (see also N. Smith),
service managers and experiment coordination value performance and sustainability.

The analysis pipeline

Centralized tools, centralized processing: ok to build software cathedrals

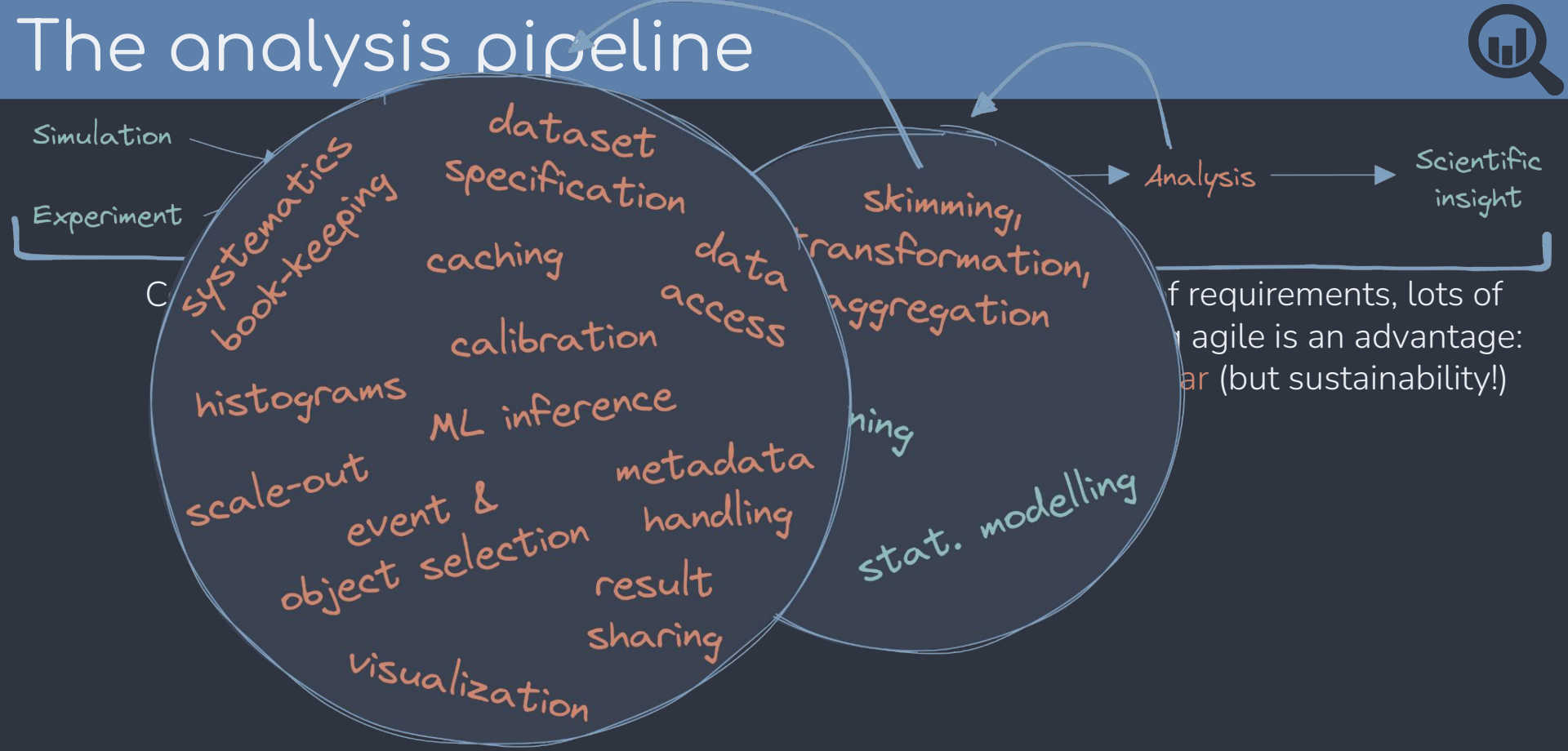
Large variety of requirements, lots of innovation, being agile is an advantage: naturally a bazaar (but sustainability!)

The analysis pipeline



Centralized tools, centralized proc … f requirements, lots of
ok to build software cathedr … agile is an advantage:
… ar (but sustainability!)

The analysis pipeline



C f requirements, lots of
agile is an advantage:
ar (but sustainability!)

The analysis pipeline

Centralized tools, centralized processing: ok to build software cathedrals

Large variety of requirements, lots of innovation, being agile is an advantage: naturally a bazaar (but sustainability!)

It's good to have **common protocols to share results and artifacts**

- data (ROOT – ideally with a simple schema)
- statistical models (common JSON schema)
- ML models (ONNX)
- complex histograms, complex large visualizations?

Making our own tools still makes sense

Reason 1: specializing for HEP needs

- hierarchical data model (event -> object -> property)
- working with collections of physics objects – efficiently
- dealing with systematic variations – efficiently
- histograms as the most common data aggregation
- ...

Making our own tools still makes sense

Reason 1: specializing for HEP needs

- hierarchical data model (event -> object -> property)
- working with collections of physics objects – efficiently
- dealing with systematic variations – efficiently
- histograms as the most common data aggregation
- …

Awkward Array

As an example, Jim Pivarski's Awkward Arrays make jaggedness (a HEP feature) pythonic (now fits in the bazaar)

Making our own tools still makes sense

Reason 1: specializing for HEP needs

- hierarchical data model (event -> object -> property)
- working with collections of physics objects – efficiently
- dealing with systematic variations – efficiently
- histograms as the most common data aggregation
- …

Awkward Array

As an example, Jim Pivarski's Awkward Arrays make jaggedness (a HEP feature) pythonic (now fits in the bazaar)

…and HEP was not the only field missing them!

Making our own tools still makes sense

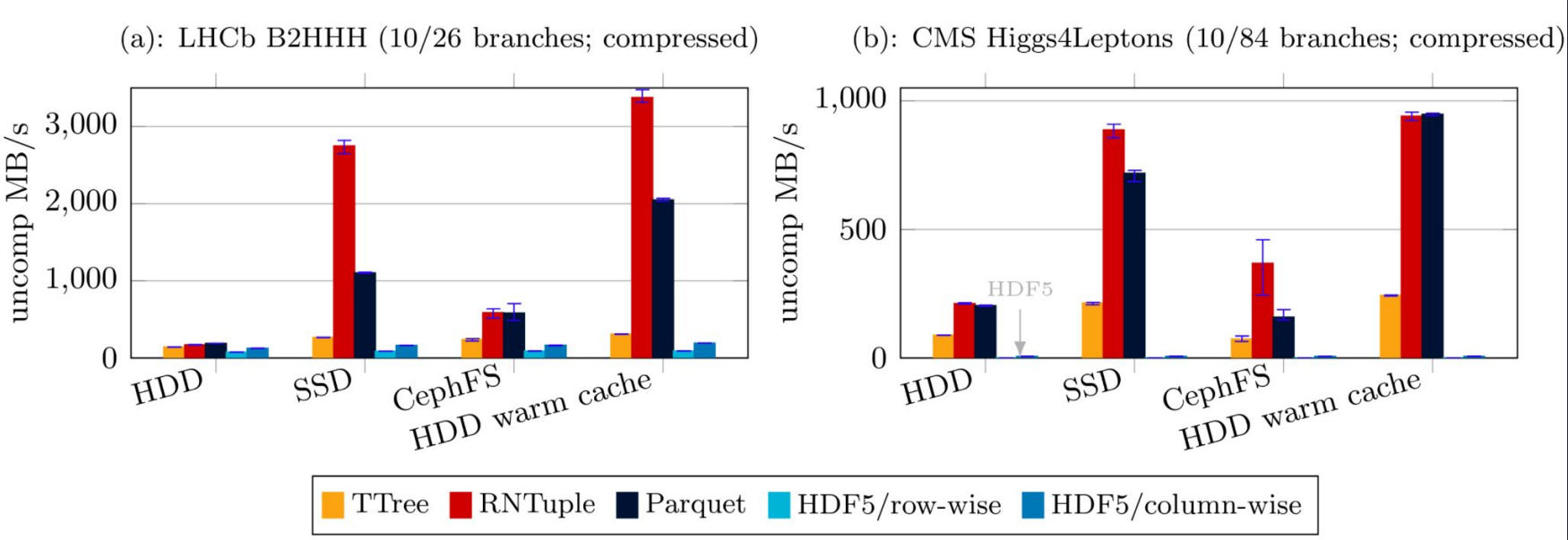
Reason 2: performance optimization

- for our I/O use cases (J Lopez, J Blomer)
- for our type of queries (D Graur, I Müller, M Proffitt et al)

Making our own tools still makes sense

Reason 2: performance optimization

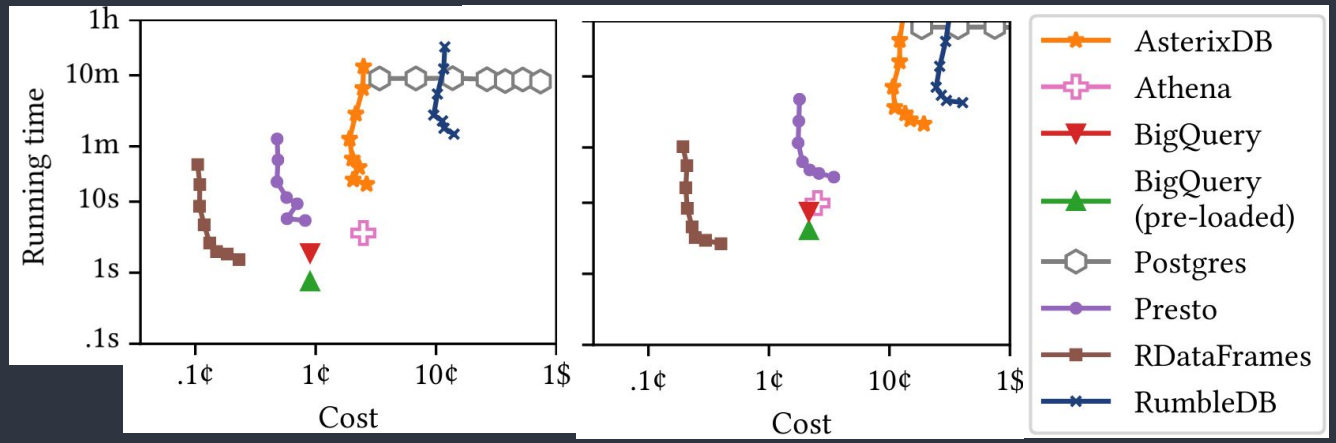
- **for our I/O use cases** (J Lopez, J Blomer)
- for our type of queries (D Graur, I Müller, M Proffitt et al)



Making our own tools still makes sense

Reason 2: performance optimization

- for our I/O use cases (J Lopez, J Blomer)
- **for our type of queries** (D Graur, I Müller, M Proffitt et al)



"[...] the general-purpose data processing systems are significantly less performant than the domain-specific ROOT framework — due to limited scalability and inefficient handling of the data and queries relevant to HEP."

Rise of the middleman analysis software

Investopedia

KEY TAKEAWAYS

- A middleman is a broker, go-between, or intermediary to a process or transaction.

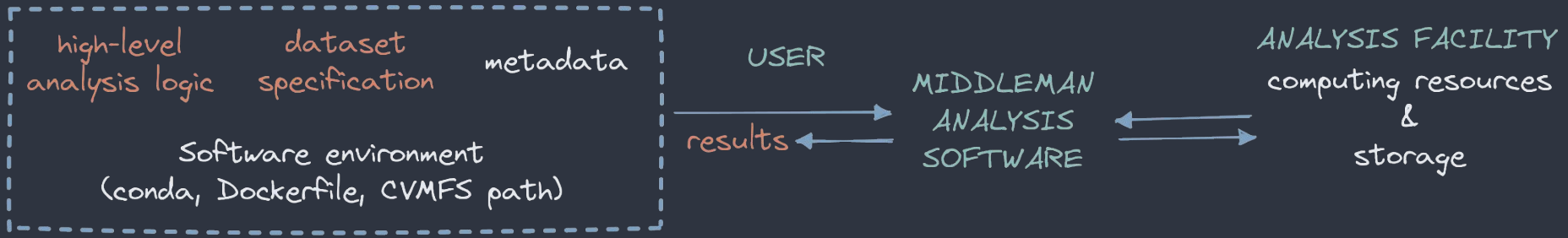
Merriam-Webster

Synonyms

broker, buffer, conciliator, go-between, honest broker, interceder, intercessor, intermediary, intermediate, interposer, mediator, peacemaker

The middleman analysis software

The middleman analysis software



Not a new concept (TTree::Draw, PROOF), but:

- current ecosystem supercharges what we can do: Python, dask or TBB schedulers, reproducible environments (e.g. conda), containerization
- we need it more than ever: hardware is more complicated (GPUs, NUMA, many-core), analysis pipelines are more complicated, performance is critical

Giving up the event loop

Responsibilities are moving upstream, from users' code or specialized frameworks to a more generic middleman layer.

Now out of users' hands, taken care of by tools such as RDataFrame, Coffea, ServiceX, bamboo, CutLang, O2.

Giving up the event loop

Responsibilities are moving upstream, from users' code or specialized frameworks to a more generic middleman layer.

Now out of users' hands, taken care of by tools such as RDataFrame, Coffea, ServiceX, bamboo, CutLang, O2.

This has a consequence on debugging experience.

How do we let users debug *their* logic without having to deal with the middleman layer?

What the middleman can do for you (1)

```
dataset_xaod = "mc15_13TeV:mc15_13TeV.361106.PowhegPythia8EvtGen_AZNLOCTEQ6L1_Zee.DAOD_STDM3"
ds = ServiceXSourceXAOD(dataset_xaod)
data = (
    ds
    .SelectMany('lambda e: (e.Jets("AntiKt4EMTopoJets"))')
    .Where('lambda j: (j.pt()/1000)>30')
    .Select('lambda j: j.pt()')
    .AsAwkwardArray(["JetPt"])
    .value()
)
```

FuncADL+ServiceX

- find xAOD file via catalog
- spin up ATLAS fwk container
- return selection result
- cache query result
- hides compute/storage load: educate users to avoid abuse

What the middleman can do for you (2)

```
1  ROOT.EnableImplicitMT() # enable multi-threading
2  h_nominal = (
3      RDataFrame('Events', 'root://eos.server/data/*.root')
4      .Vary('Muon_pt', 'RVec<RVecF>{0.9*Muon_pt, 1.1*Muon_pt}', ['down', 'up'])
5      .Filter('nMuon == 2 && Muon_charge[0] != Muon_charge[1]')
6      .Define('mass', 'InvariantMass(Muon_pt, Muon_eta, Muon_phi, Muon_mass)')
7      .Histo1D('mass')
8  )
9  # dictionary with keys 'nominal', 'Muon_pt:down', 'Muon_pt:up'
10 h_dict = ROOT.RDF.VariationsFor(h_nominal)
```

ROOT.RDataFrame

- transparent multi-threading
- simple systematics
- seamless scale-out

What the middleman can do for you (3)

Advantages beyond ergonomics and performance

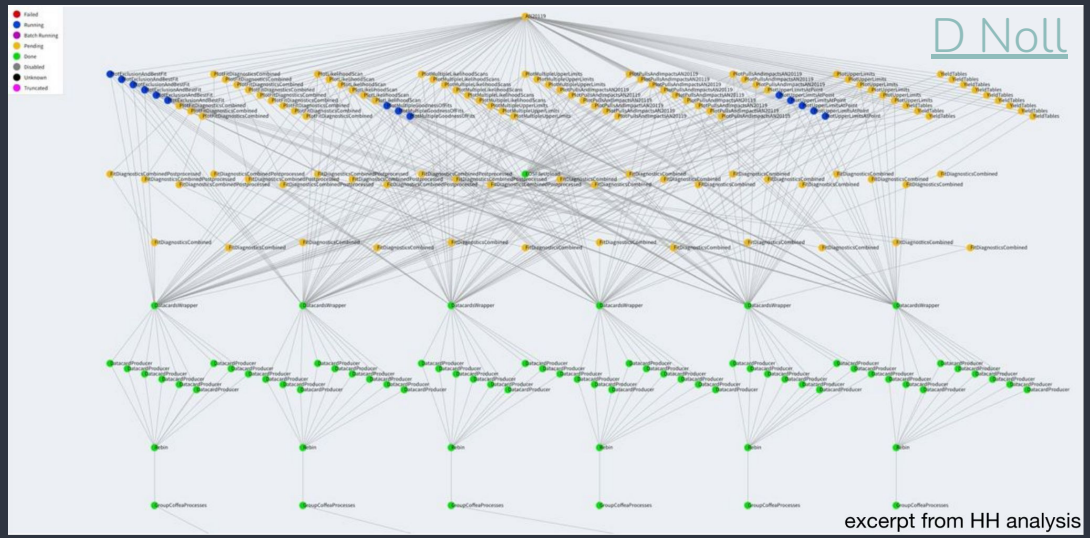
- enhanced reproducibility on changing hardware and infrastructure
- transparent caching into object stores (V Padulano et al)
- simpler comparison of different analyses
- GPU offloading (e.g. of ML inference)
- transparent caching of ML inference results?
- automated analysis preservation?
- ...?

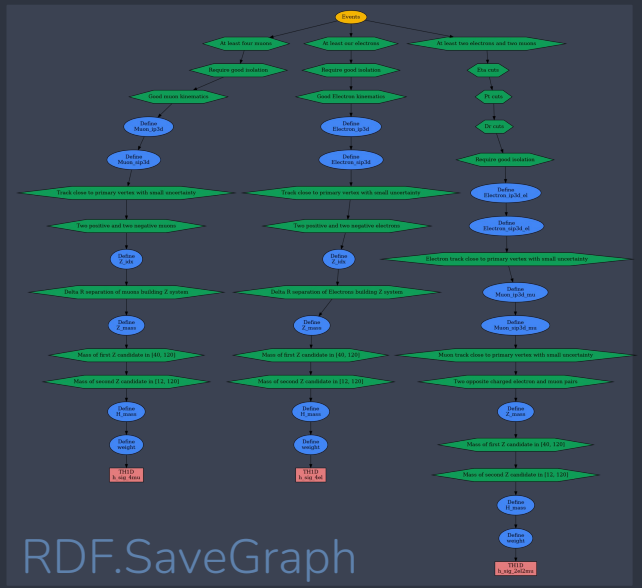
A few
recurring implementation details

Two levels of computation graphs

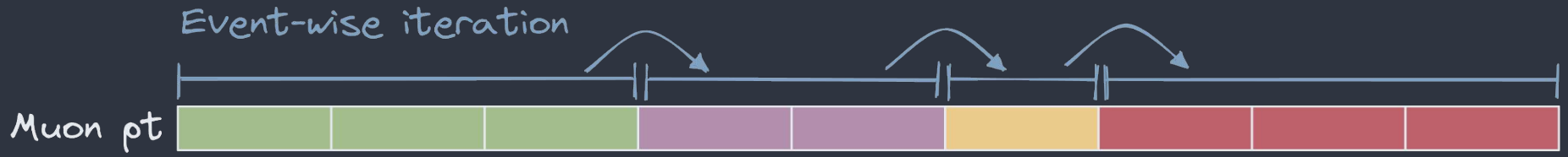
At the analysis logic level (awkward+dask, RDF, bamboo, Gandiva (O2)), and at the analysis workflow level (Snakemake, Luigi, law).

A programmatic handle on the operations to perform and their dependencies: good for workflow optimization, caching, potentially auto-differentiation.

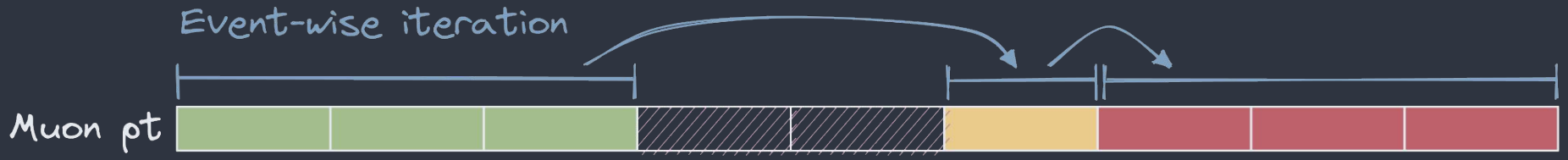




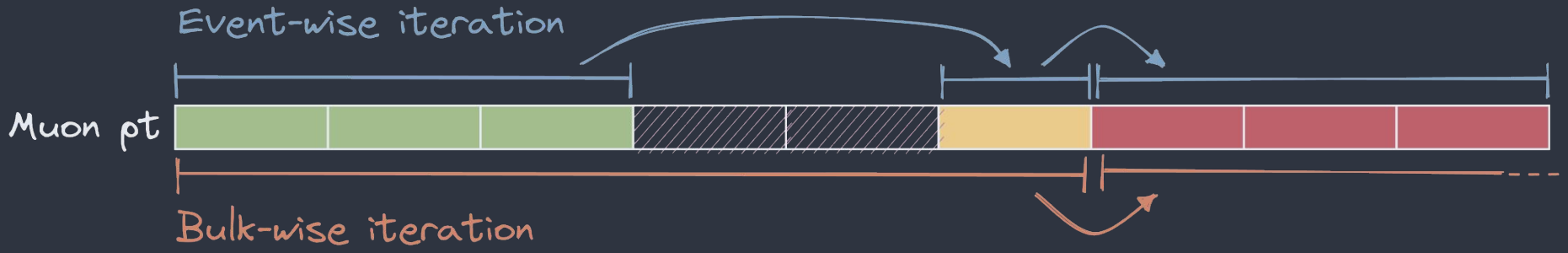
Event-wise, bulk-wise logic



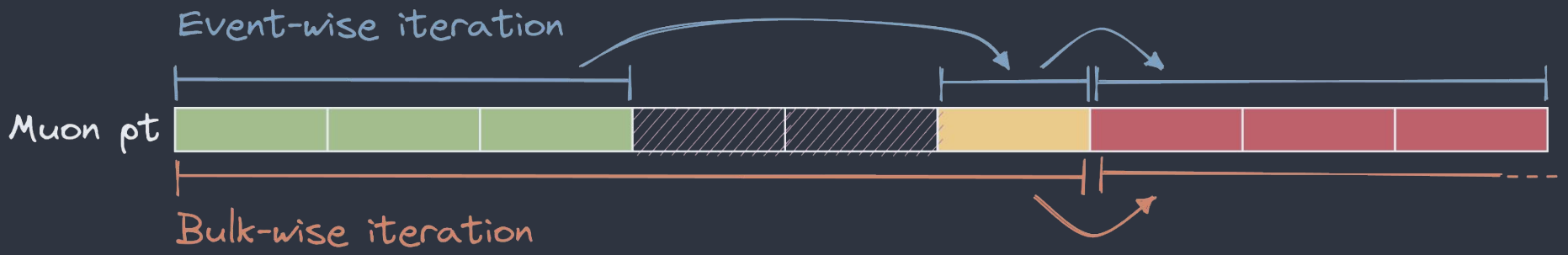
Event-wise, bulk-wise logic



Event-wise, bulk-wise logic

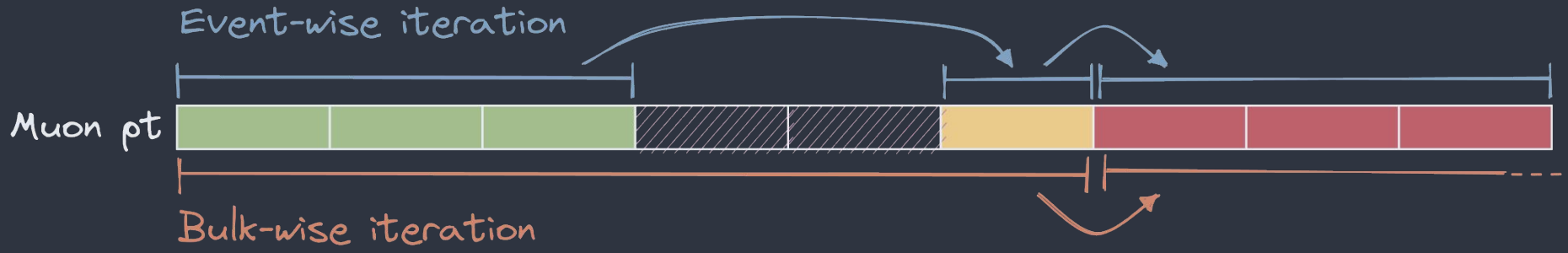


Event-wise, bulk-wise logic



- think vector operations on Numpy arrays vs for loops on C++ vectors
- bulk-wise required in pure Python for performance, with per-bulk operations
- handling multiple events is sometimes cumbersome
  -> Numba functions can be used to go back to explicit for loops
- can be useful in C++ interfaces in some cases e.g. to enable GPU offloading

Event-wise, bulk-wise logic

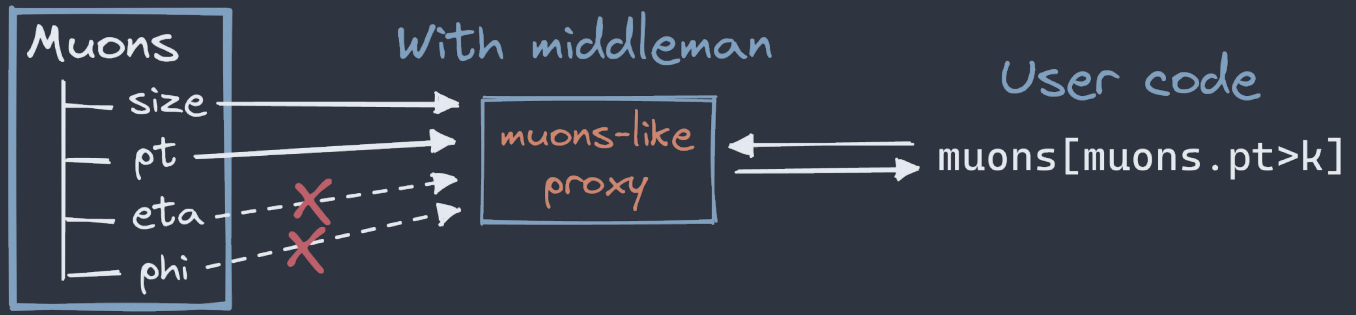


- think vector operations on Numpy arrays vs for loops on C++ vectors
- bulk-wise required in pure Python for performance, with per-bulk operations
- handling multiple events is sometimes cumbersome
  -> Numba functions can be used to go back to explicit for loops
- can be useful in C++ interfaces in some cases e.g. to enable GPU offloading

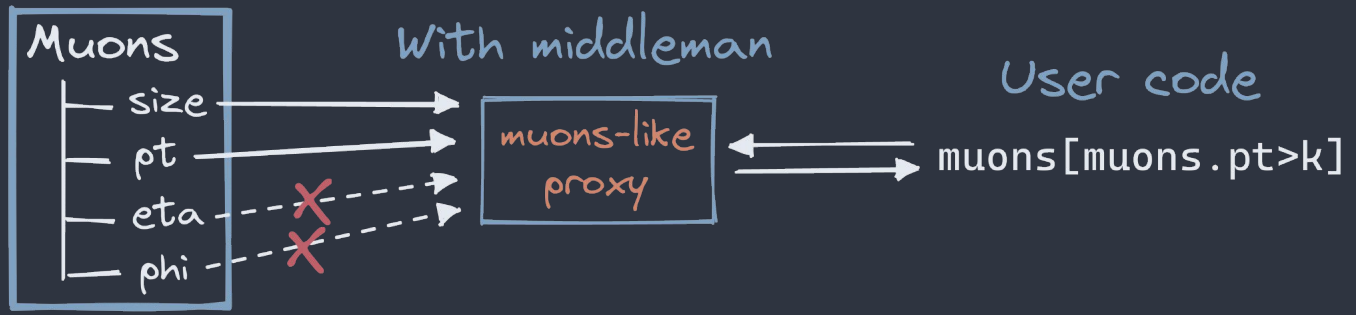
Bulk-wise APIs do not automatically imply better CPU vectorization, because of event/object masks introducing branching: an interesting optimization opportunity?

Efficient object collections

Efficient object collections

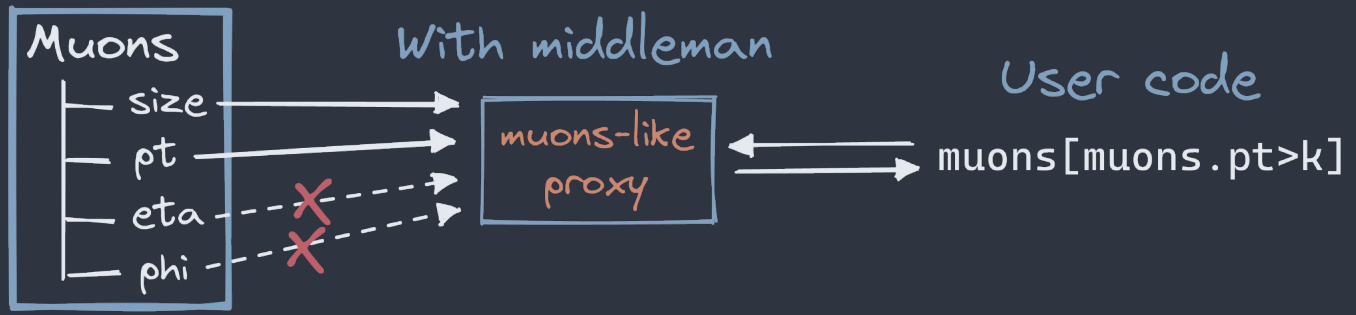


Efficient object collections



Exposing objects to analysts can be more easily decoupled from I/O operations.
This reduces the amount of data read dramatically.

Efficient object collections



Exposing objects to analysts can be more easily decoupled from I/O operations.
This reduces the amount of data read dramatically.

Proxies in Coffea (bulk-wise)

```
events.Jet[abs(events.Jet.eta) < 1].pt
```

from the Coffea ADL benchmarks

Proxies in bamboo (event-wise)

```
op.select(tree.Jet, lambda j : op.abs(j.eta) < 1.)
```

from the bamboo ADL benchmarks

Building analysis facilities for the bazaar

What do you mean “analysis facility”?

To work out you go to the gym, to build cool things you go to a makerspace, to work on your analysis you connect to an analysis facility (AF).

What do you mean “analysis facility”?

To work out you go to the gym, to build cool things you go to a makerspace, to work on your analysis you connect to an analysis facility (AF).

Things that an AF manager and a gym coach can both say

- “Let me show you how to use that machine”
- “Here’s a simple program to get you started”
- “I think you are loading the wrong weights”

What do you mean “analysis facility”?

To work out you go to the gym, to build cool things you go to a makerspace, to work on your analysis you connect to an analysis facility (AF).

To an extent, LXBATCH is a an AF, but **we can make things more comfortable**.

Common building blocks

- containerization (Docker/Singularity, Kubernetes)
- dask as a scheduler, often in tandem with HTCondor/SLURM
- JupyterLab as frontend (SSH access also allowed)
- high-bandwidth connection to storage
- dedicated resources vs “parasitic” usage of existing ones?

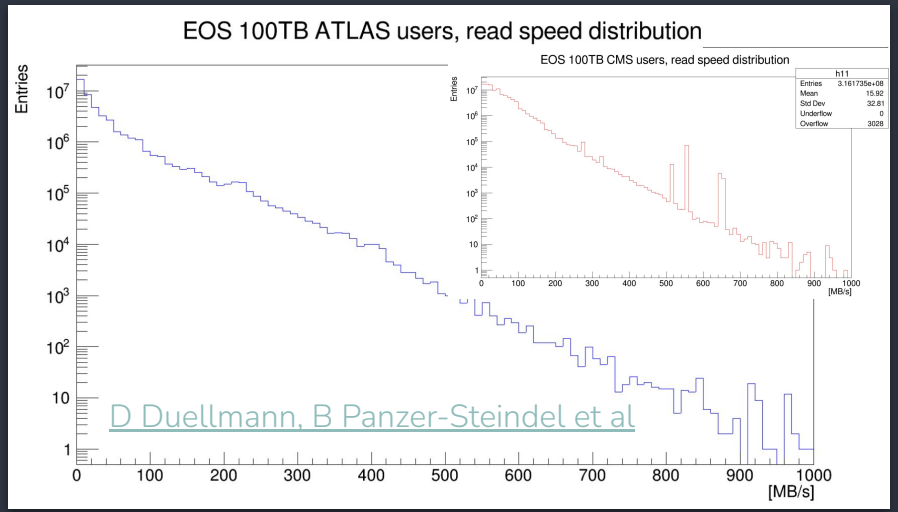
See also the Second Analysis Ecosystem Workshop Report.

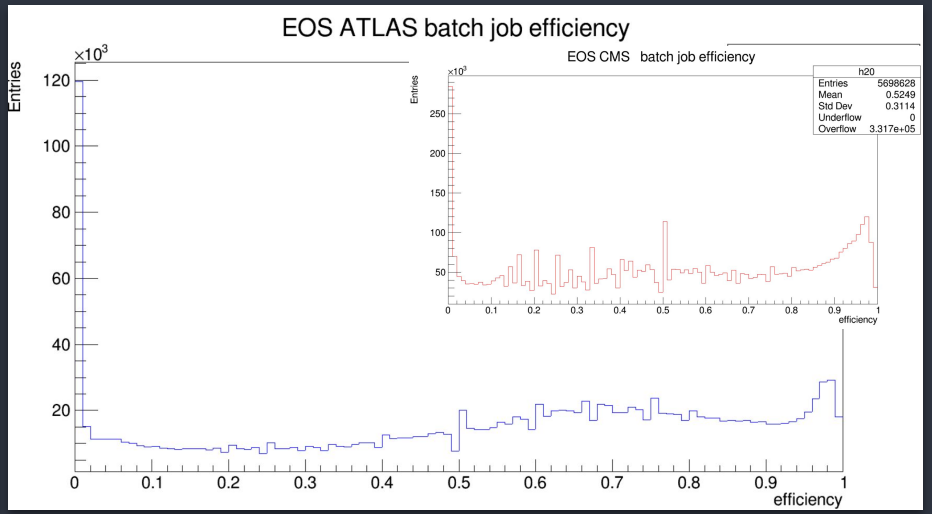
What the analysis facility can do for you

- simplify experiment authentication and data access
- smart scheduling to guarantee less cache thrashing
- monitoring:
  - feedback to users (“you have throughput 100x lower than the median”)
  - feedback to developers (“latency too high for remote data fetching”)

What the analysis facility can do for you

- simplify experiment authentication and data access
- smart scheduling to guarantee less cache thrashing
- monitoring:
  - feedback to users (“you have throughput 100x lower than the median”)
  - feedback to developers (“latency too high for remote data fetching”)





Analysis facilities for the bazaar

Optimize for common patterns and behaviors, not for specific software stacks.

Build monitoring so that users and admins can compare solutions.

Analysis facilities for the bazaar

Optimize for common patterns and behaviors, not for specific software stacks.

Build monitoring so that users and admins can compare solutions.

Example: built-in semantic distinction between

- quick exploration: low latency (interactive), small data, can use small, fast caches
- full analysis: high throughput on big data, higher latency is ok, might benefit from train-like scheduling to use larger caches well

ALICE AFs already have this concept (test runs on 10% data before full runs).

Conclusions

Some of the current challenges

We are not done, of course. We are learning how to do better in several areas:

1. Flexibility: hide complexity without losing flexibility, e.g by
   - design customization points, or
   - let users move to the lower layer (GUI -> YAML -> Python –> C++)

Some of the current challenges

We are not done, of course. We are learning how to do better in several areas:

1. Flexibility: hide complexity without losing flexibility, e.g by
   - design customization points, or
   - let users move to the lower layer (GUI -> YAML -> Python –> C++)
2. Debugging: let users debug logical and performance issues in the tip of the iceberg without having to understand the full iceberg

Some of the current challenges

We are not done, of course. We are learning how to do better in several areas:

1. Flexibility: hide complexity without losing flexibility, e.g by
   - design customization points, or
   - let users move to the lower layer (GUI -> YAML -> Python –> C++)
2. Debugging: let users debug logical and performance issues in the tip of the iceberg without having to understand the full iceberg
3. Caching: smart caching required to meet performance goals; requires collaboration between analysis, schedulers, facilities

Some of the current challenges

We are not done, of course. We are learning how to do better in several areas:

1. Flexibility: hide complexity without losing flexibility, e.g by
   - design customization points, or
   - let users move to the lower layer (GUI -> YAML -> Python –> C++)
2. Debugging: let users debug logical and performance issues in the tip of the iceberg without having to understand the full iceberg
3. Caching: smart caching required to meet performance goals; requires collaboration between analysis, schedulers, facilities
4. Derived datasets: prevent blow-up, e.g. by making it simple to join analysis-specific observables with centrally-produced datasets

Some of the current challenges

We are not done, of course. We are learning how to do better in several areas:

1. Flexibility: hide complexity without losing flexibility, e.g by
   - design customization points, or
   - let users move to the lower layer (GUI -> YAML -> Python –> C++)
2. Debugging: let users debug logical and performance issues in the tip of the iceberg without having to understand the full iceberg
3. Caching: smart caching required to meet performance goals; requires collaboration between analysis, schedulers, facilities
4. Derived datasets: prevent blow-up, e.g. by making it simple to join analysis-specific observables with centrally-produced datasets
5. Heterogeneous computing: offload appropriate computations to GPUs

Analysis is getting simpler *and* faster

The HEP analysis software ecosystem is **healthy** and evolving at a fast pace with a good mix of R&D and production-grade developments.

Analysis is getting simpler *and* faster

The HEP analysis software ecosystem is **healthy** and evolving at a fast pace with a good mix of R&D and production-grade developments.

The LHC Run 3 era will benefit from a new generation of analysis tools that focus on gathering semantic information about the analysis (input, environment, code, ...) and HEP-specific concepts (systematics, physics objects, ...).

Analysis is getting simpler *and* faster

The HEP analysis software ecosystem is **healthy** and evolving at a fast pace with a good mix of R&D and production-grade developments.

The LHC Run 3 era will benefit from a new generation of analysis tools that focus on gathering semantic information about the analysis (input, environment, code, ...) and HEP-specific concepts (systematics, physics objects, ...).

Happy coding!

Backup

Things with disruptive potential

Things that may disrupt the ecosystem *if they gain traction* (10-year scale):

- julia: C++ perf., Python ergonomics, but low adoption, large migration effort
- end-to-end automatic differentiation: advantages still unclear, would require the collaboration of large parts of the analysis software stack

What about GPUs?

GPUs, TPUs and similar accelerators can speed up parts of the analysis pipeline (ML training/inference, PDF evaluation, maybe appropriate parts of the data processing), and they can fit in the existing paradigms.

What about quantum computers?

Similar to GPUs: quantum computers are the “ultimate accelerators” but also extremely specialized, large input datasets might be problematic.

Is your analysis "fast"?

There are legitimate use cases where a throughput of O(10) evts/s is optimal.

However, here are some examples of what is possible today
(and things are only getting better):

- "turnaround of a few hours for [...] thousands of histograms of the CMS Run 2 data on a batch system", P David
- 3.2B events, O(1000) systematics, 70 5-dim histograms in 45 minutes (SSD storage, 128 threads) J Bendavid
- NanoAOD events processed at 400 kHz when producing ~6k histograms (SSD storage, 128 threads), E Manca, E Guiraud
- Events processed at ~20 kHz/core when running the Analysis Grand Challenge on a Coffea-casa analysis facility (network read, 400 cores), A Held, O Shadura

RNTuple and other advancements should provide another factor N speed-up (1<N<10).

Performance targets

1 PB of (compressed) data, of which 100 TB are actually read by the analysis.
We expect the analysis will be able to run in A. 10 minutes on a cluster of 64 nodes, or B. 4 hours on a single beefy machine with 128 cores.

Throughput required: A. ~3 GB/s/node or B. ~100 MB/s/core for read+processing.

- need hardware setup that can sustain such throughput
- cannot afford reading more than what's strictly needed
- must make good use of the hierarchy of storage options
  - remote
  - large shared storage at the level of the computing facility (xcache, high-bandwidth object stores)
  - small user-level storage