# Physics Analysis use-cases and GCP

Nikolai Hartmann[1], Lukas Heinrich[2]

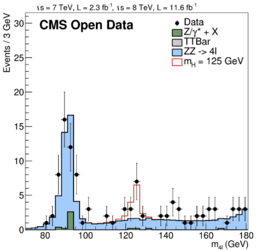
[1]LMU Munich, [2]CERN

February 17, 2021, ATLAS - Google Technical Meeting

# An impressive demo

Lukas Heinrich and Ricardo Rocha at the KubeCon 2019 → youtube recording, chep talk

**70 TB** Dataset → Cluster on GKE → Job Results → Interactive Visualization →




Max **25000 Cores**

Single Region, 3 Zones

25000 **Kubernetes Jobs**

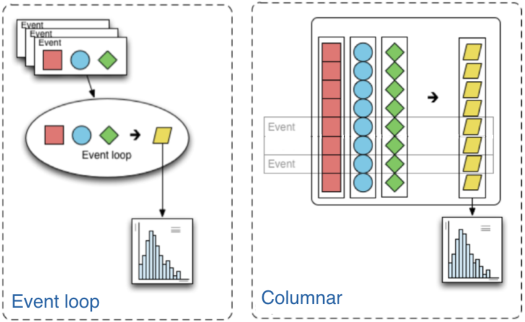
Aggregation

**Reperform the Higgs discovery analysis on 70 TB of CMS open data in a live demo**

# What to do next?

**The demo was using “toy” data and analysis software from 2010**
$\rightarrow$ Can we use this for realistic datasets?
$\rightarrow$ How does this look like with modern tools?

# Columnar data analysis




- Traditional analysis workflow in HEP involves processing one event at a time
- Columnar (array-at-a-time) processing in python is becoming standard in data science
  → thanks to tools like numpy, tensorflow, etc
  → becoming increasingly popular in HEP as well
- Lot's of progress in recent years in the HEP python ecosystem
  → e.g uproot, awkward array and coffea

**A larger scale columnar data analysis could be a nice fit for GCP!**

# Dataset and example analysis

- 100 TB dataset with ATLAS LHC Run2 data in derived format
  $\rightarrow$ DAOD_PHYSLITE: small analysis format, calibrations applied
- Distributed across 260k files, 18e9 events in total
- Stored in ROOT format, columns split
  $\rightarrow$ potential for conversion to parquet
- Example analysis using uproot and awkward array:
  - Apply selection criteria for analysis objects: Electrons, Muons, Jets
  - Perform overlap removal (involves combinatorics)
  - Can then calculate simple observables, fill histograms
  - Reads $\approx$ 10% of the stored data
    $\rightarrow$ but rather scattered reading: basket (compressed block) sizes in the order of 5-50kb
  - Maximum throughput when reading from memory: 10k events per second
    (still mostly dominated by decompression/deserialization)

# Scaling

Want to try 2 approaches:

**Via PanDa (ATLAS workflow managment system)**

- Experience from previous Google project
- Submit as user job script (prun)

**Via Dask**

- Directly submit tasks as python futures or high-level constructs like distributed data frames
- Running dask on kubernetes in Google cloud probably (?) straight forward
- Good for interactive analysis, e.g. using Jupyter notebooks
- Already collected some experience from running $\approx$ 100 workers
  $\rightarrow$ Up to which point can we scale this?

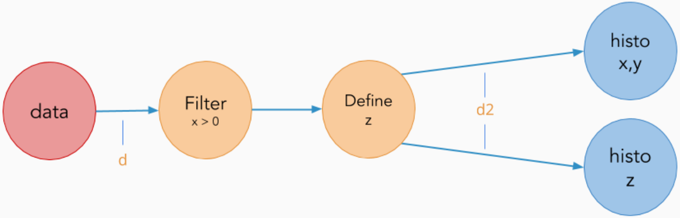
# Parquet, RNtuple and other alternative storage formats

- Current ROOT format suboptimal for columnar reading
  $\rightarrow$ lot's of overhead
  $\rightarrow$ too small baskets (compression units)
  $\rightarrow$ columnwise storage only one level of nesting deep
  (only event-wise byte offsets stored separately)
- Two attractive formats to address this:
  $\rightarrow$ Apache Parquet: Already quite mature, works well with awkward array
  $\rightarrow$ RNtuple: Future ROOT format, still under development
- Another alternative: "Pure" columnar storage
  $\rightarrow$ Don't distinguish indices/offsets and data on storage level
  (supported by awkward array)
  $\rightarrow$ Can use almost any format that allows storing and retrieving columns (e.g. HDF5, npz)

**Plan: Try at least with Parquet on Google cloud**

# RDataFrame

Framework for declarative analysis (part of ROOT)




```cpp
// d2 is a new data-frame, a transformed version of d
auto d2 = d.Filter("x > 0")
           .Define("z","x*x + y*y");
// make histograms out of it
auto hz = d2.Histo1D("z");
auto hxy = d2.Histo2D({"hxy","hxy",16,-1,1,64,-1,1},"x","y");
```

→ Demonstrator analysis on top of spark cluster: https://cds.cern.ch/record/2655457
(based on Google Summer of Code project PyRDF)

# Questions

- How to read the data?
  Uproot supports http(s), local/posix file access, memory mapped files
- When reading ROOT files via http, multipart/byteranges might be interesting
  (but depends on latency - maybe not so important for low latency)