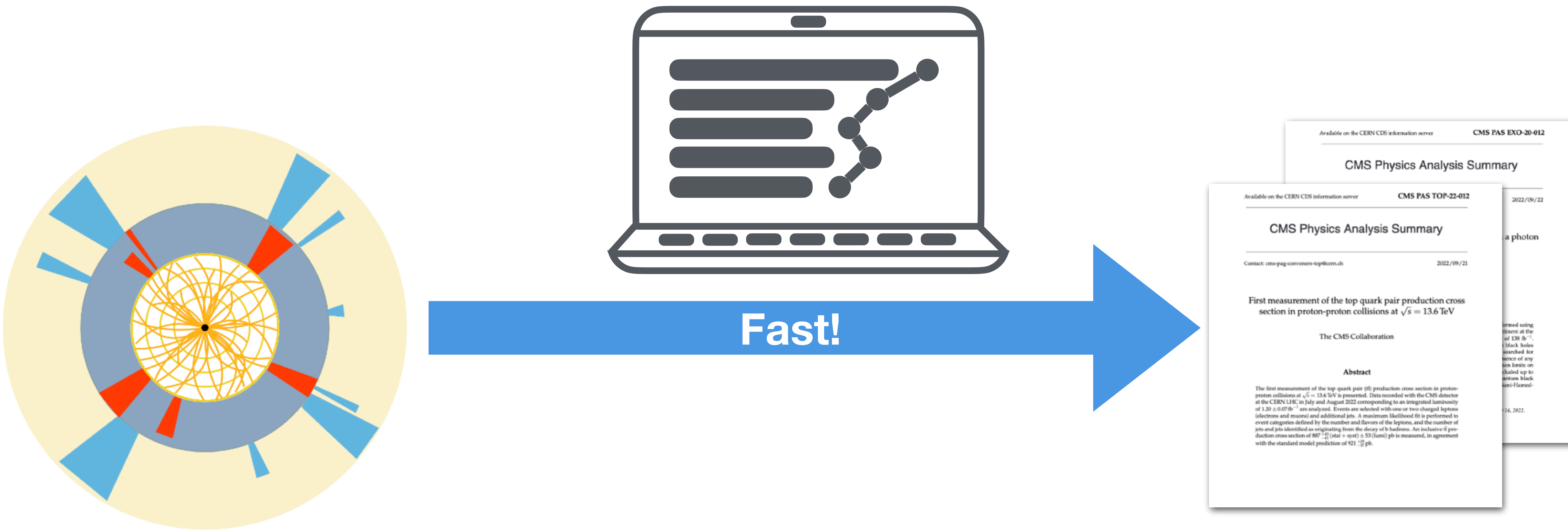


Fast Columnar Physics Analyses of Terabyte-Scale LHC Data on a Cache-Aware Dask Cluster (2207.08598)

Niclas Eich, Martin Erdmann, Peter Fackeldey, Benjamin Fischer, Dennis Noll, Yannik Rath

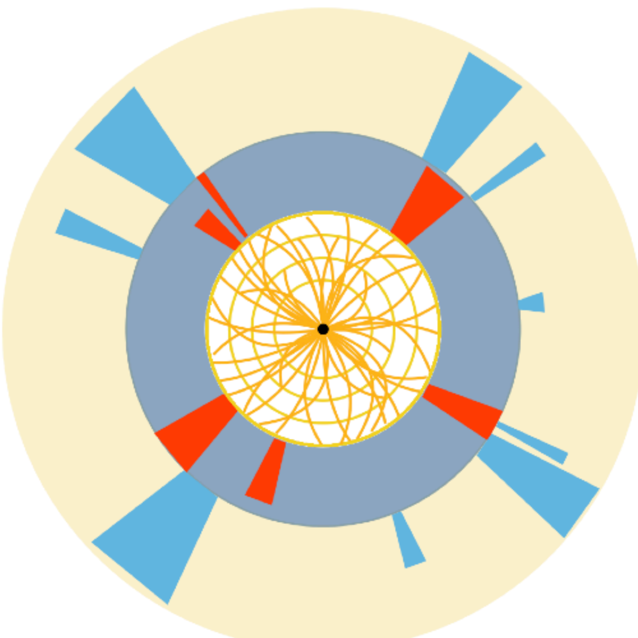
IRIS-HEP Topical Meeting

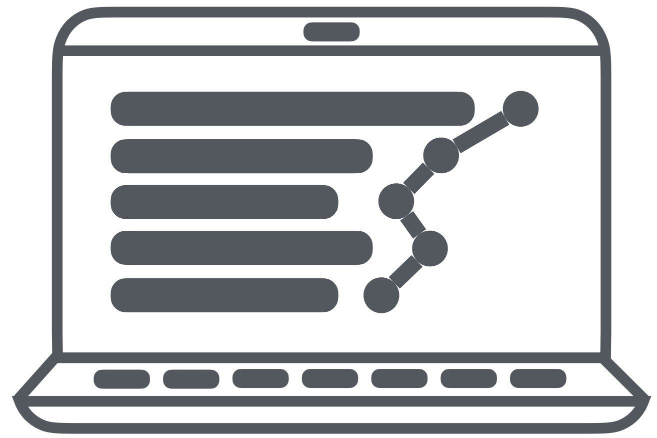
17.10.22

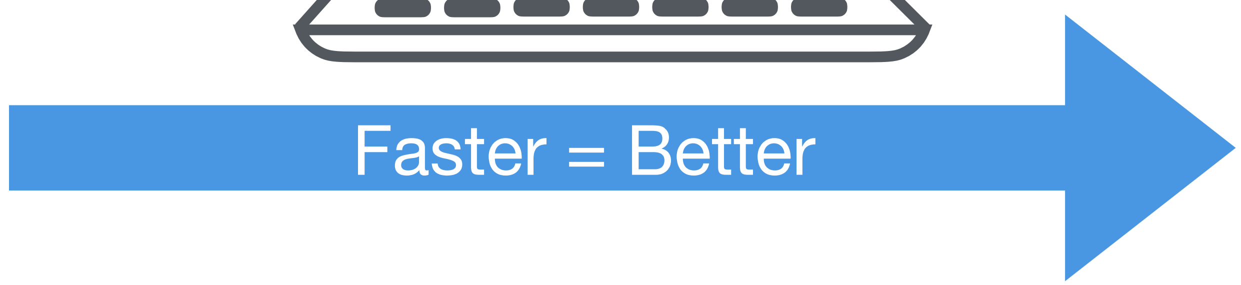
RWTH AACHEN UNIVERSITY

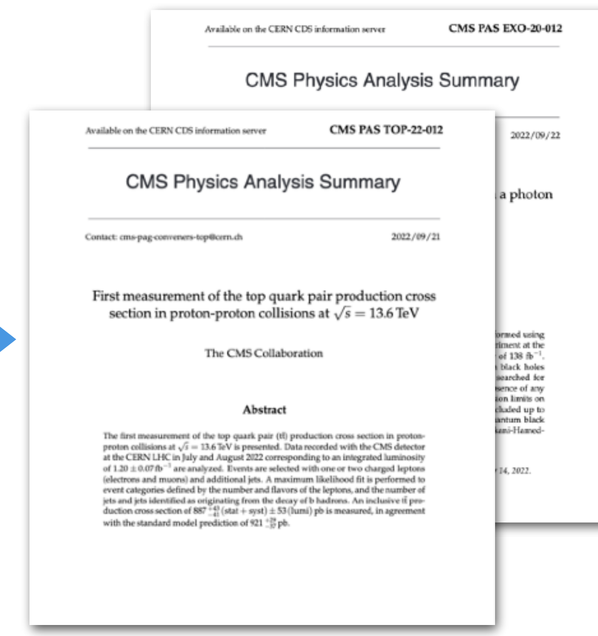
Motivation

Input	**Analysis**	**Output**
Data ~ O(TB)		Results ~ O(kB)



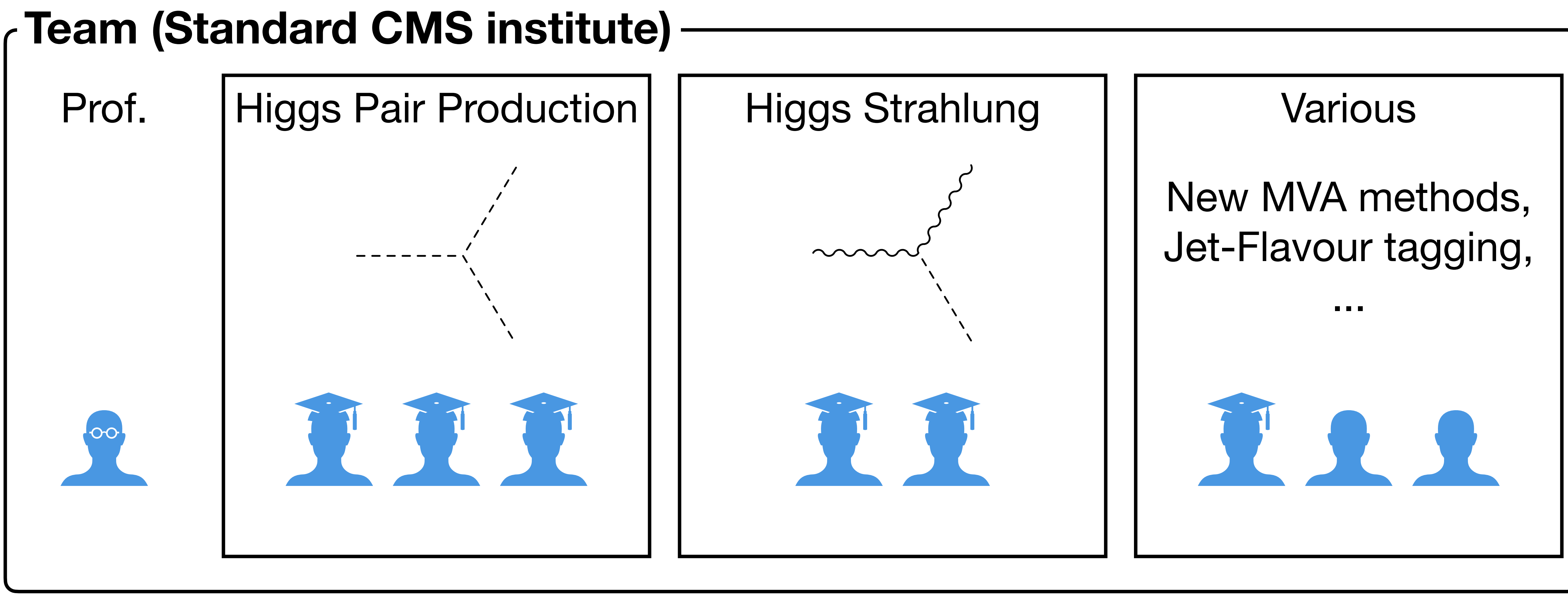






- Small time-to-insight drives high physics potential:
  - Tuning of existing methods
  - Investigations of new methods
- Ultimate goal: Analysis in duration of a coffee break
- Also: Crucial for HL LHC analyses

Physics Analysis & Team



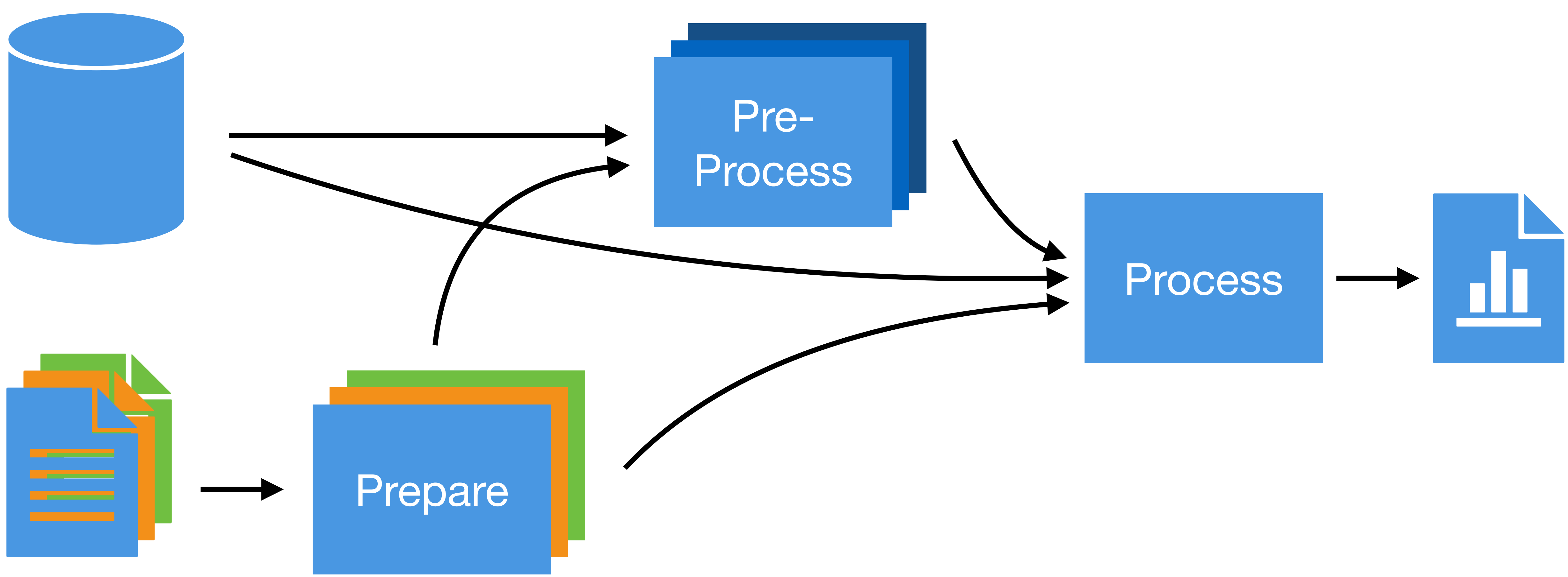
- Analyses have common challenges:
  - Many inputs:
    - Simulations and recorded data
    - Meta data (Efficiency factors, Corrections, ...)
  - Heavy computations (Event reconstruction, MVA algorithms, ...)
  - Bookkeeping
- <u>Shared software repository</u>

Analysis Structure (1/2)

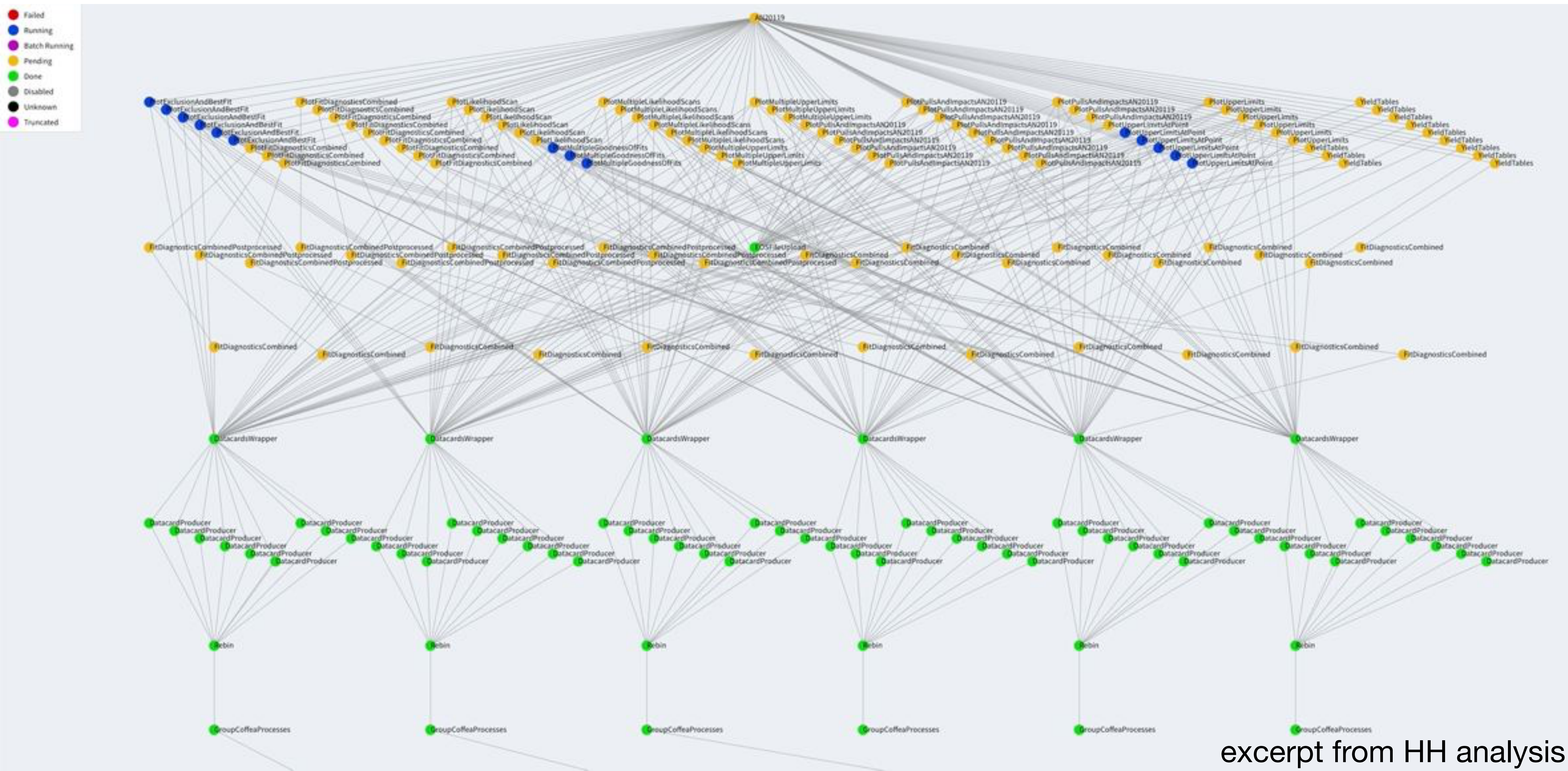
- Inputs: Event data, meta data
- O(100) different computing steps:
  - Preparation of meta data
  - Pre-processing of event data
  - Main-processing of event data
- Outputs: Histograms, plots, ...
- Tasks connected using law

Typical workflows

- Full analysis
- Development (repeated)
- Debugging (repeated)
- Plotting features of data



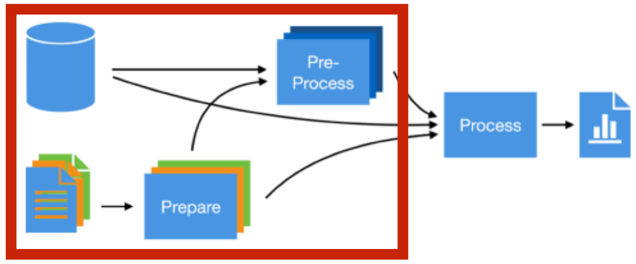
Analysis Structure (2/2)



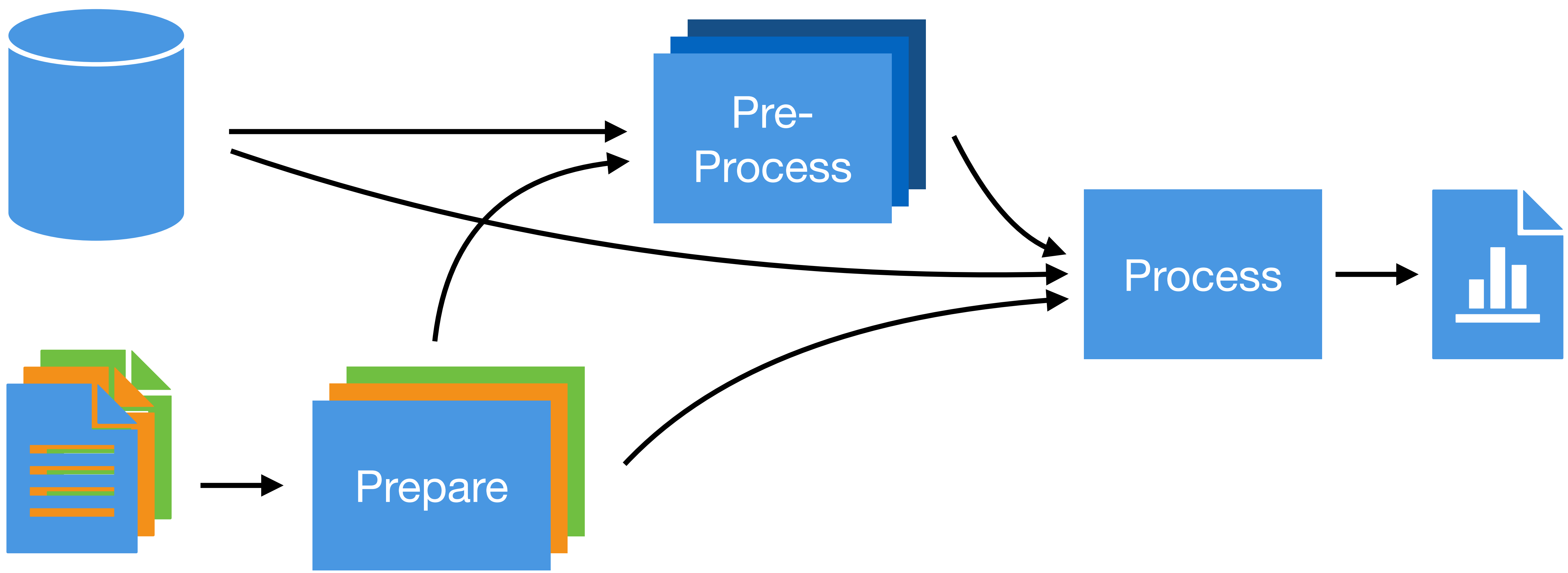
excerpt from HH analysis

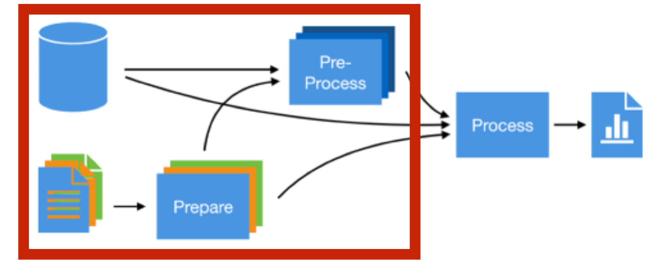
- Make-like execution of whole analysis (`law run FullAnalysis`)
- Visual task graph representation using Luigi Scheduler
  - Used for overview of run status, structural improvements, debugging

Analysis Structure - Inputs

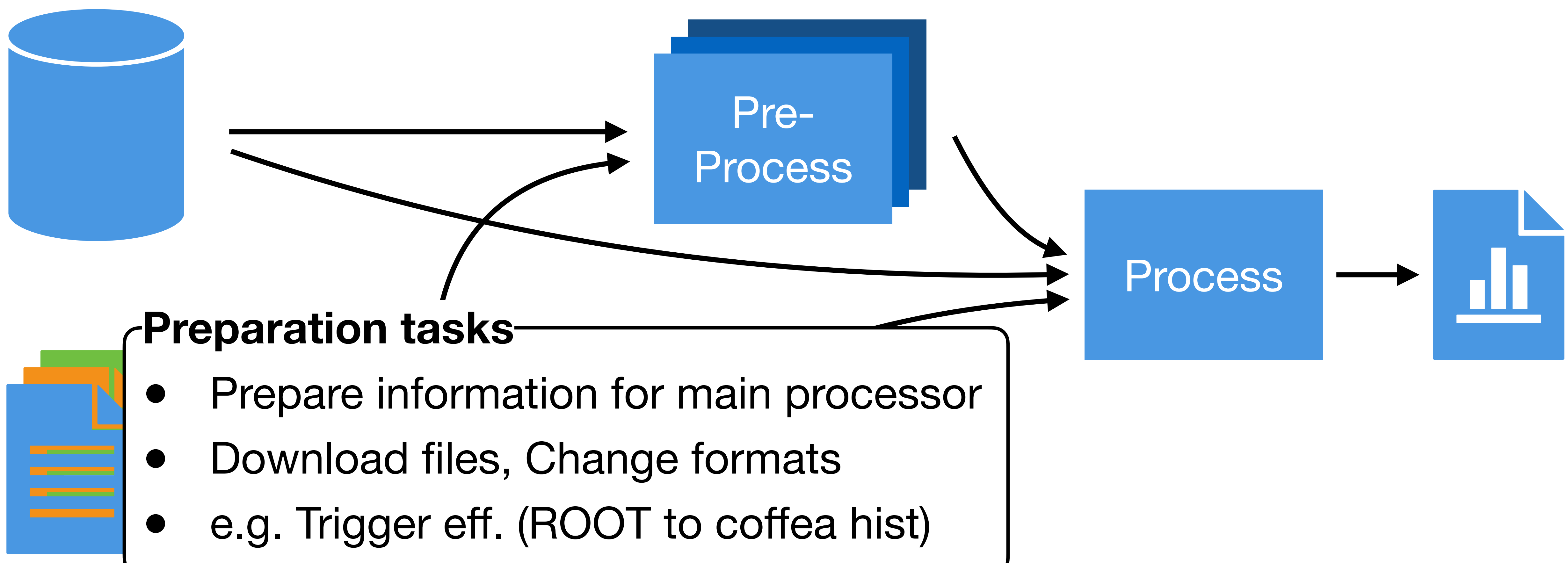


- Different kinds of inputs:
  - Event data O(10TB):
    - Recorded data & Simulation
    - NanoAOD format ~1kB per event
  - Meta data O(MB):
    - Efficiency measurements, scaling factors, ...
    - Twiki pages, JSON files, Custom ROOT files, ...





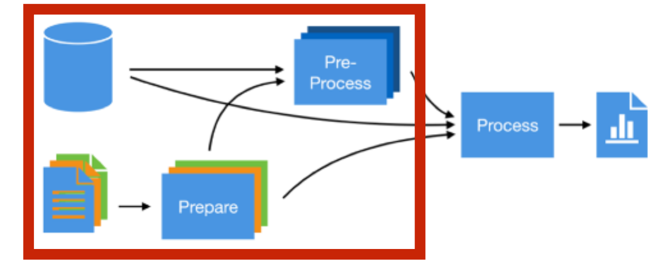
- Different kinds of inputs:
  - Event data O(10TB):
    - Recorded data & Simulation
    - NanoAOD format ~1kB per event
  - Meta data O(MB):
    - Efficiency measurements, scaling factors, ...
    - Twiki pages, JSON files, Custom ROOT files, ...



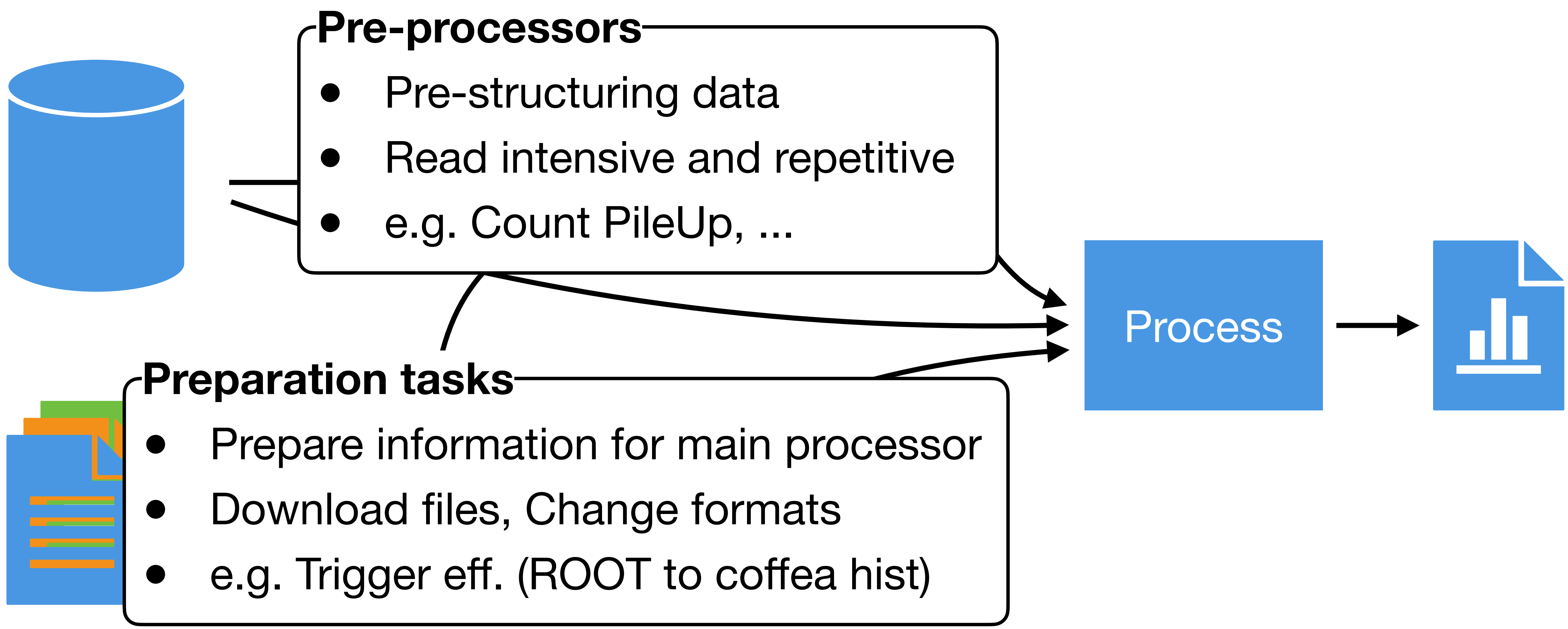
Preparation tasks

- Prepare information for main processor
- Download files, Change formats
- e.g. Trigger eff. (ROOT to coffea hist)

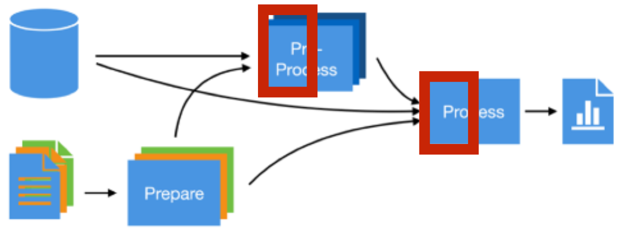
Dennis Noll - 17.10.22



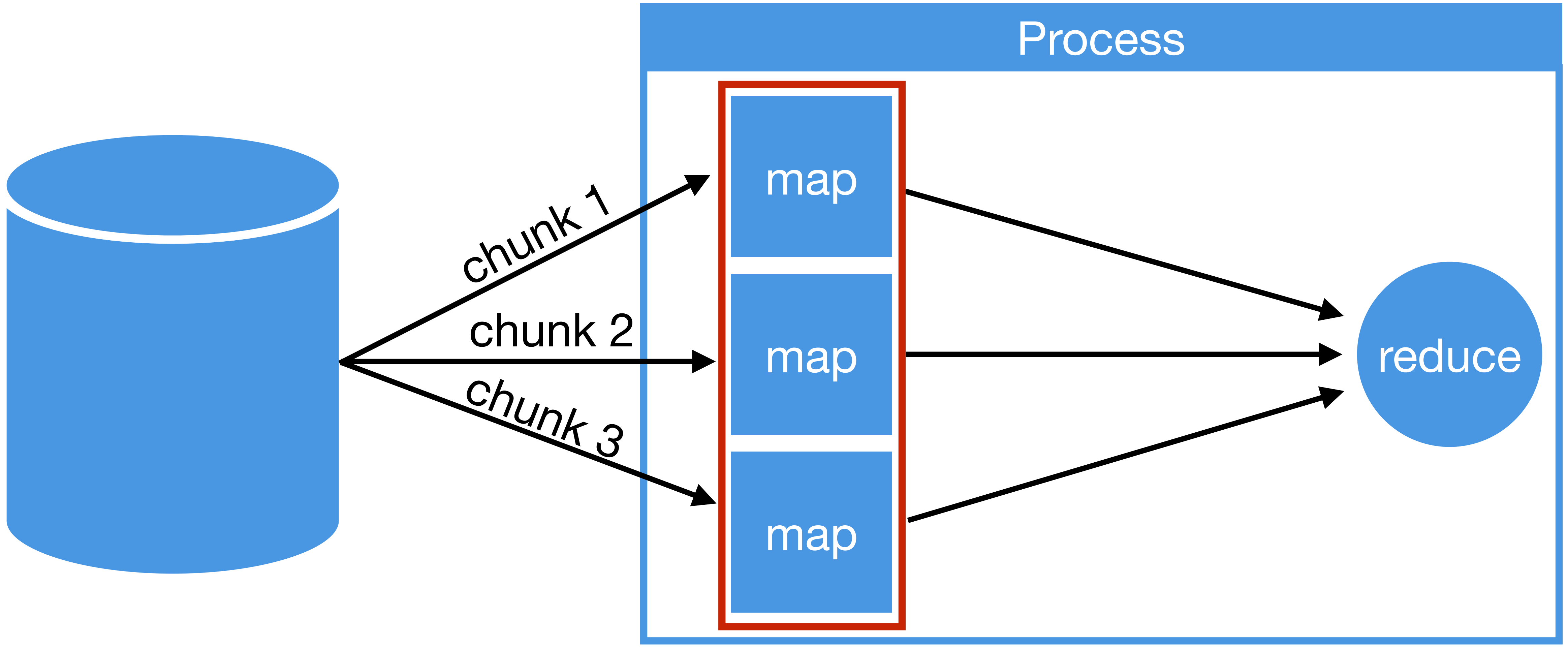
- Different kinds of inputs:
  - Event data O(10TB):
    - Recorded data & Simulation
    - NanoAOD format ~1kB per event
  - Meta data O(MB):
    - Efficiency measurements, scaling factors, ...
    - Twiki pages, JSON files, Custom ROOT files, ...

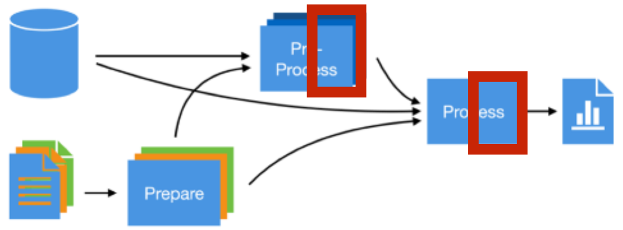


Analysis Structure - Processor Map

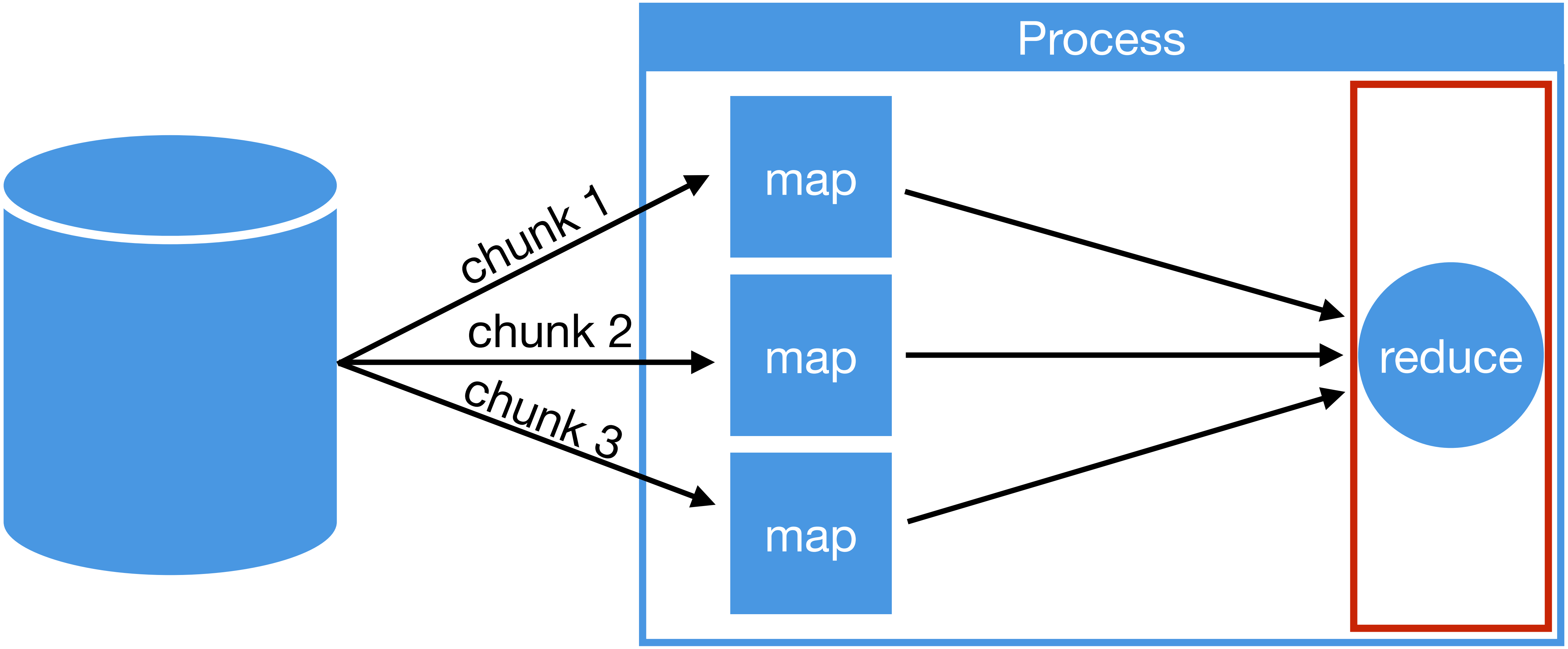


- Processor *maps* perform real computing payload
- Fast:
  - Vectorised processing (`awkward`, `numpy`)
  - Central GPU server for MVA evaluation
- Parallel processing in two levels of hierarchy:
  - Datasets (ttbar, ST, DY, ...)
  - Chunks (100k events)



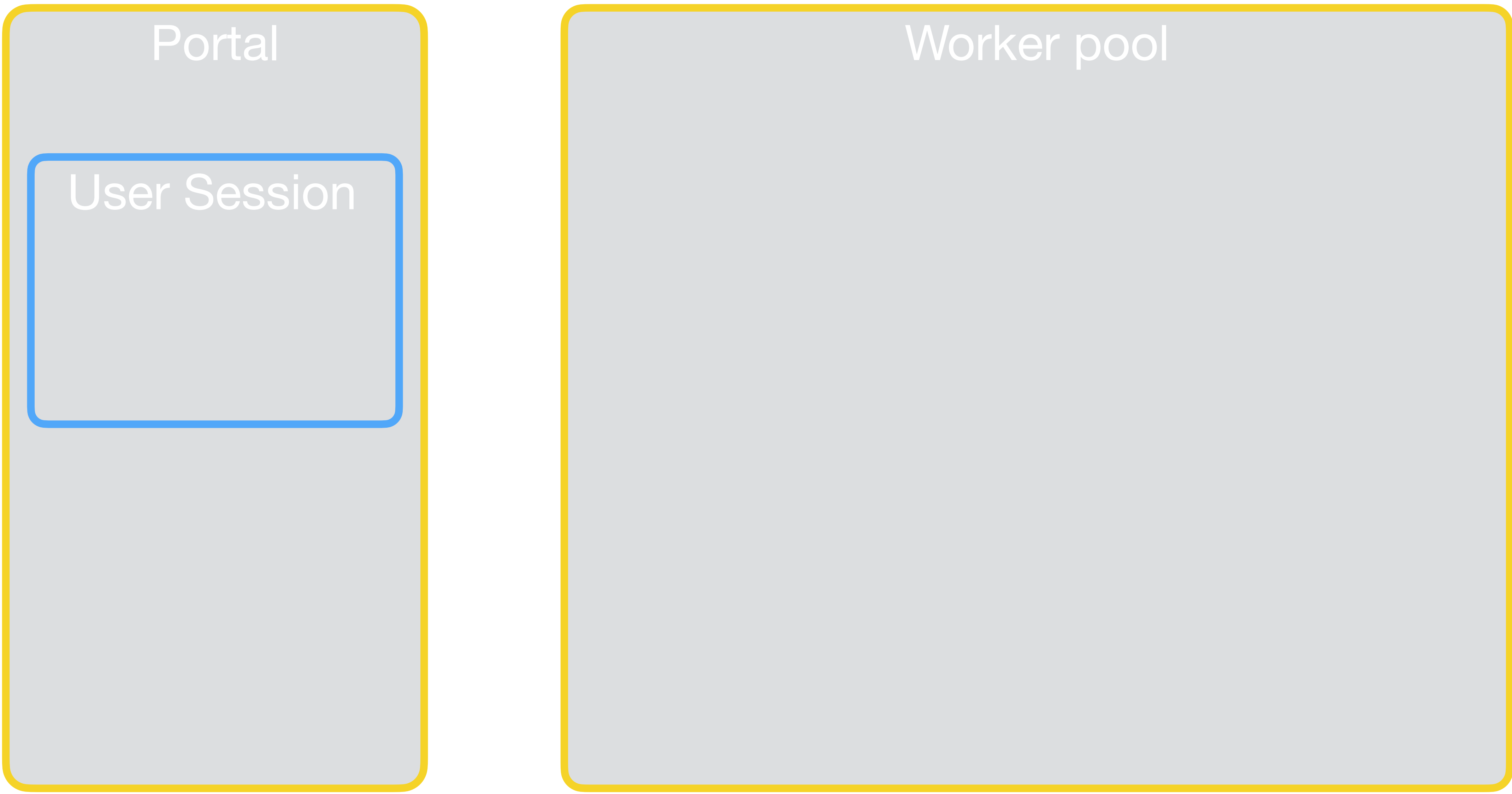


- *Reduces* objects returned by map steps:
  - Histograms, event counts, cutflow statistics, ...
- Histograms are special case:
  - Can be multidimensional and very large O(GB)
  - Implemented fast histogramming methods (here):
    - `hist` with `h5py` backend
    - e.g. Expand, merge, compression, ...



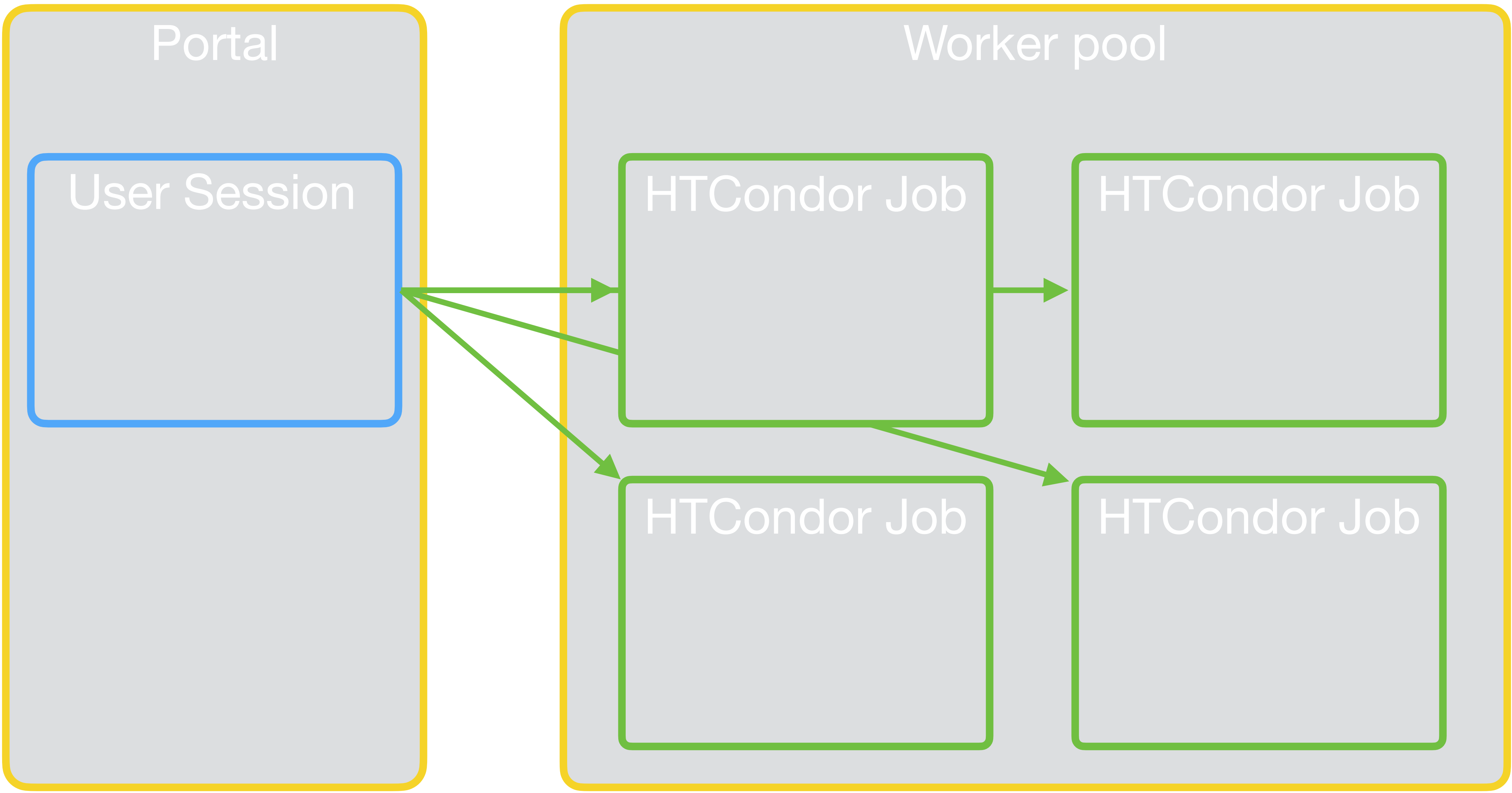
Map Scale Out: HTCondor + Dask

- *Maps* can be parallelised over multiple worker nodes
- Using HPC cluster with one portal node and worker pool
- Two step process:



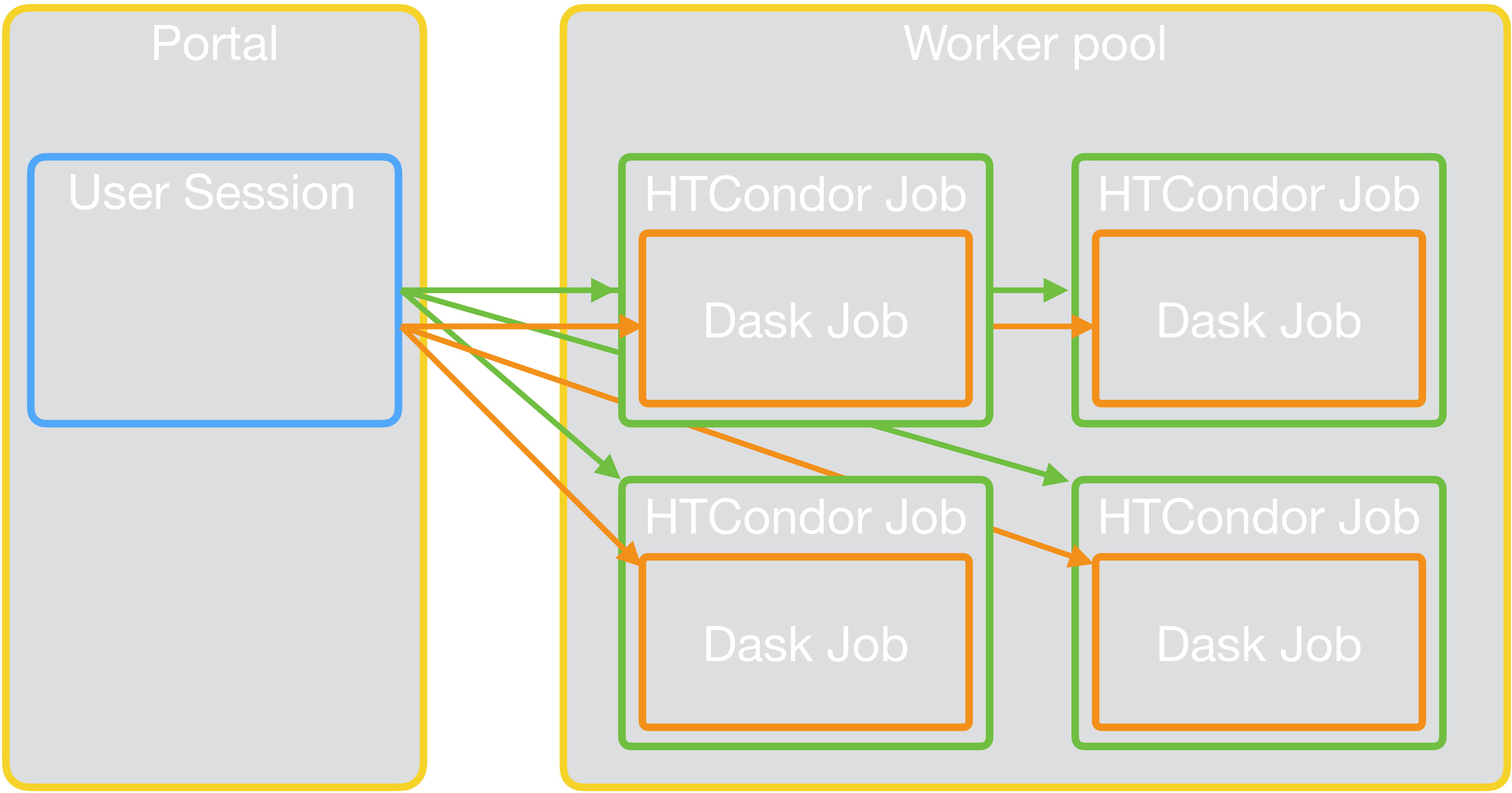
Map Scale Out: HTCondor + Dask

- *Maps* can be parallelised over multiple worker nodes
- Using HPC cluster with one portal node and worker pool
- Two step process:
  1. HTCondor opens jobs in worker pool (1 CPU Thread, 1.5 GB RAM)



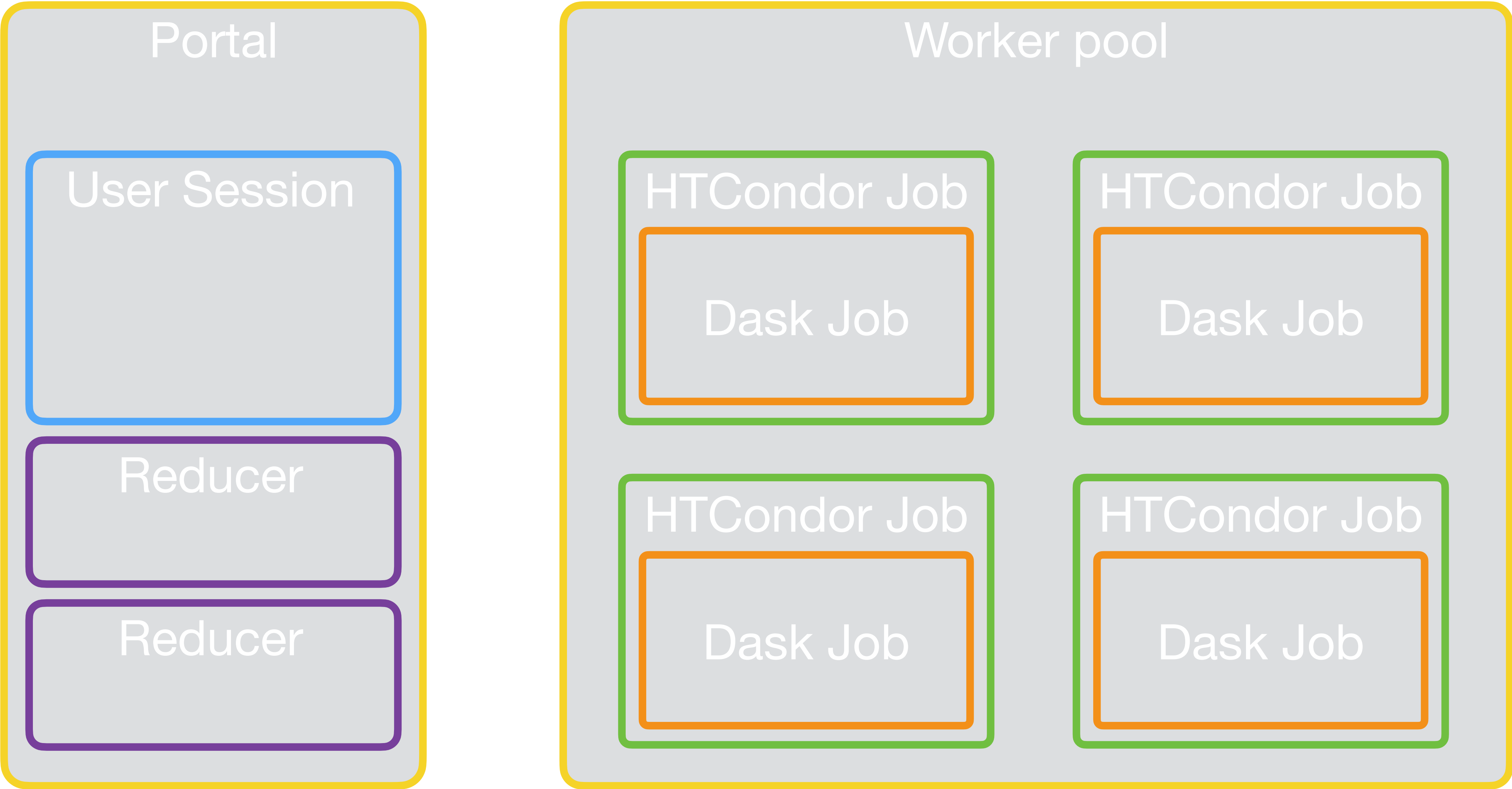
Map Scale Out: HTCondor + Dask

- *Maps* can be parallelised over multiple worker nodes
- Using HPC cluster with one portal node and worker pool
- Two step process:
    1. HTCondor opens jobs in worker pool (1 CPU Thread, 1.5 GB RAM)
    2. Dask uses HTCondor slots



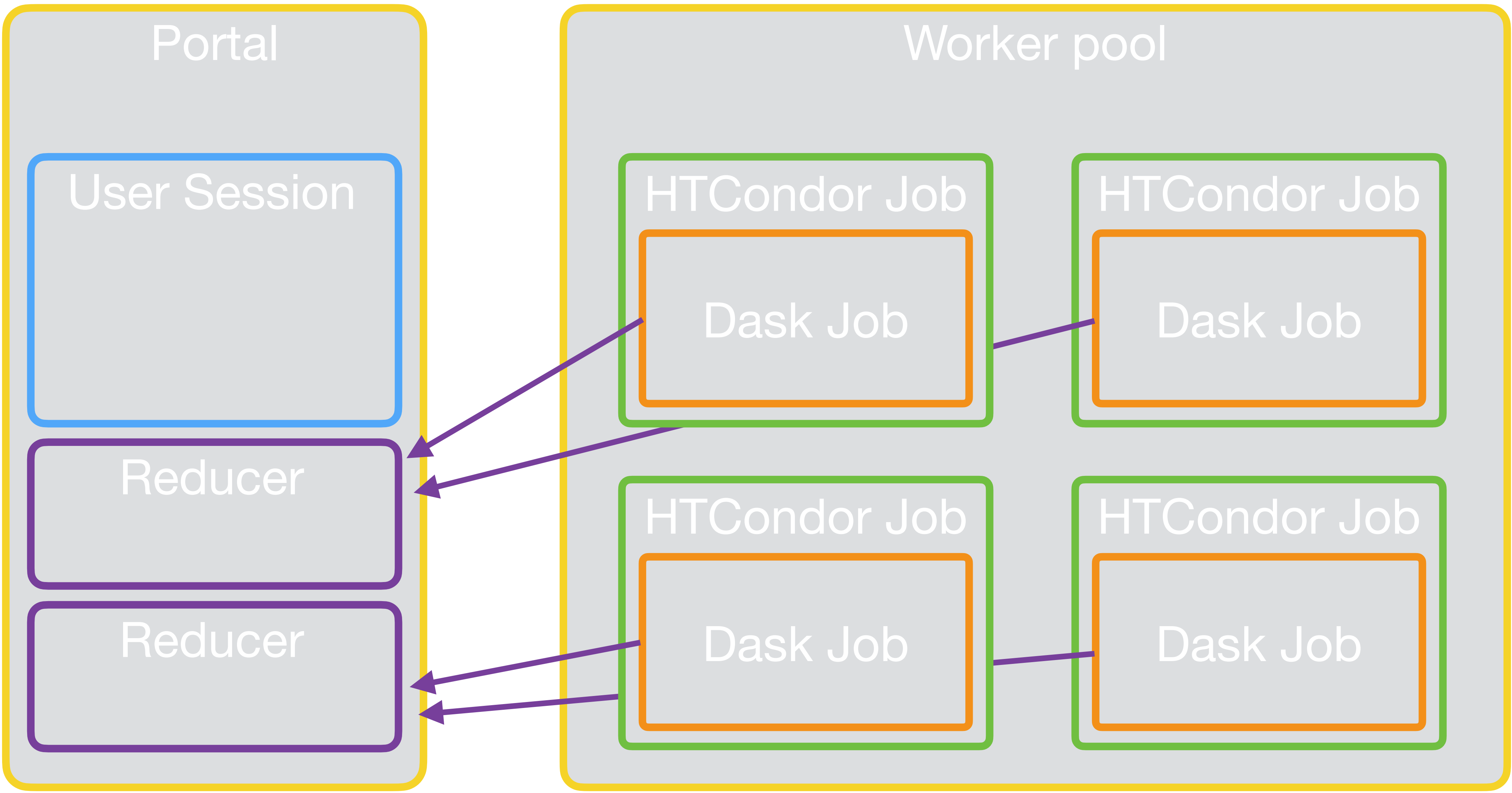
Map Scale Out: Reducing

- Outputs of parallel map jobs (e.g. histograms) must be *reduced*
- Ten parallel reducers on portal node
- Each reducer has two reducing instances:
  - Normal reduce: map jobs finished regularly, reduce job pulls
  - Early reduce: map jobs need space, map job pushes



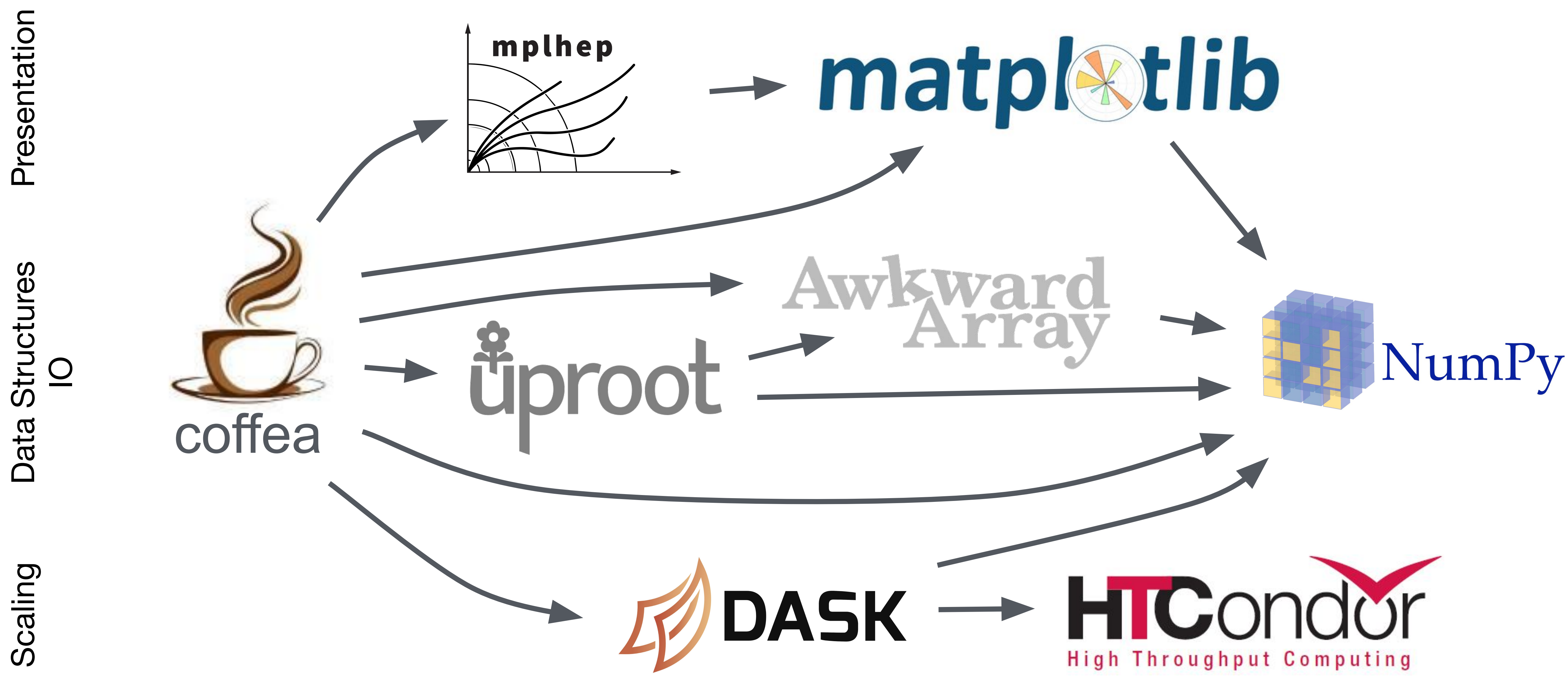
Map Scale Out: Reducing

- Outputs of parallel map jobs (e.g. histograms) must be *reduced*
- Ten parallel reducers on portal node
- Each reducer has two reducing instances:
  - Normal reduce: map jobs finished regularly, reduce job pulls
  - Early reduce: map jobs need space, map job pushes



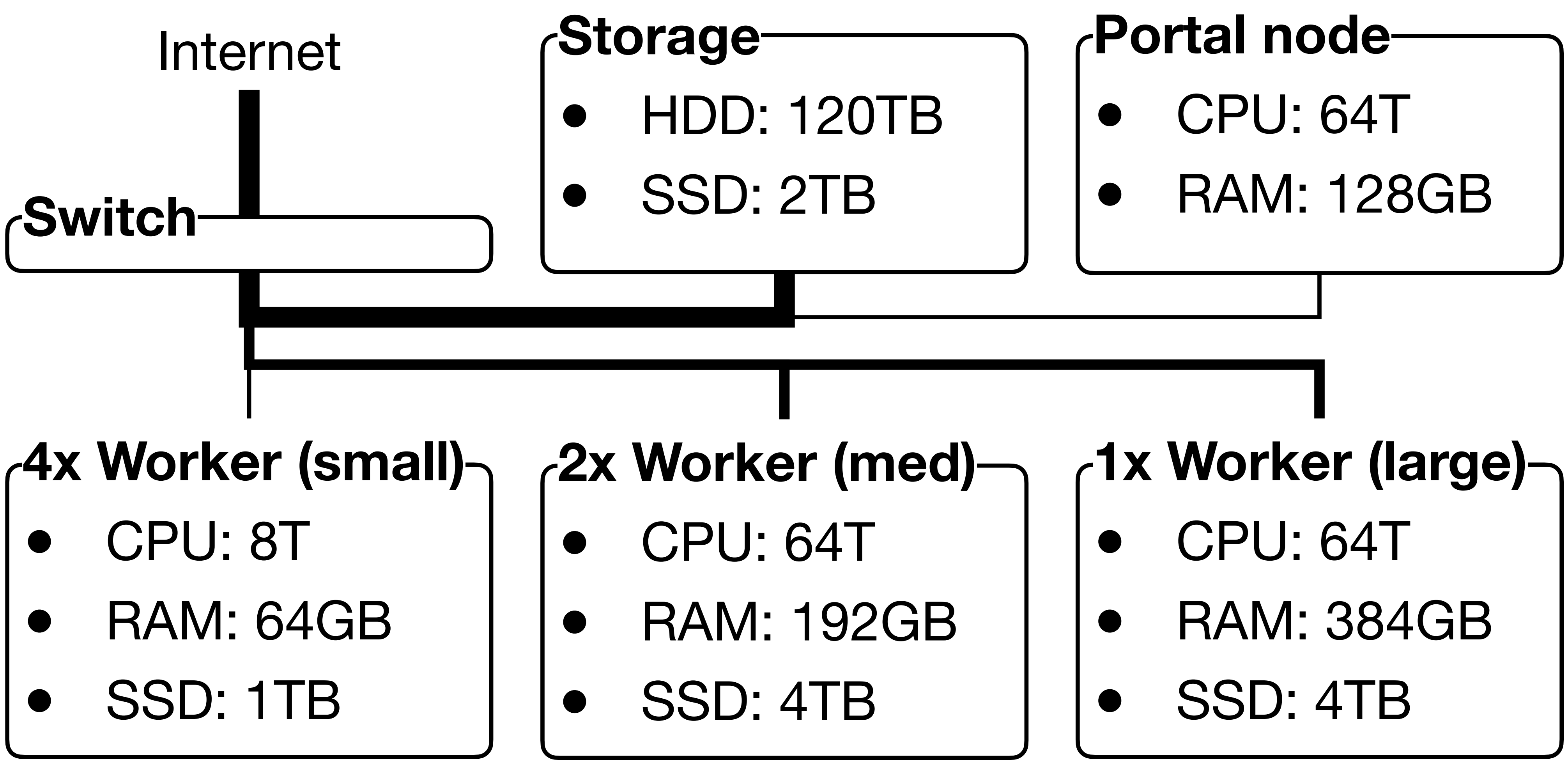
Software Stack

- Everything implemented in **python** (analysis repository)
- Modular software framework:
  - Standard tools: NumPy, matplotlib, Dask, HTCondor, ...
  - HEP specific tools: AwkwardArray, uproot, mplhep, coffea, ...



Resource: VISPA Cluster (1/2)

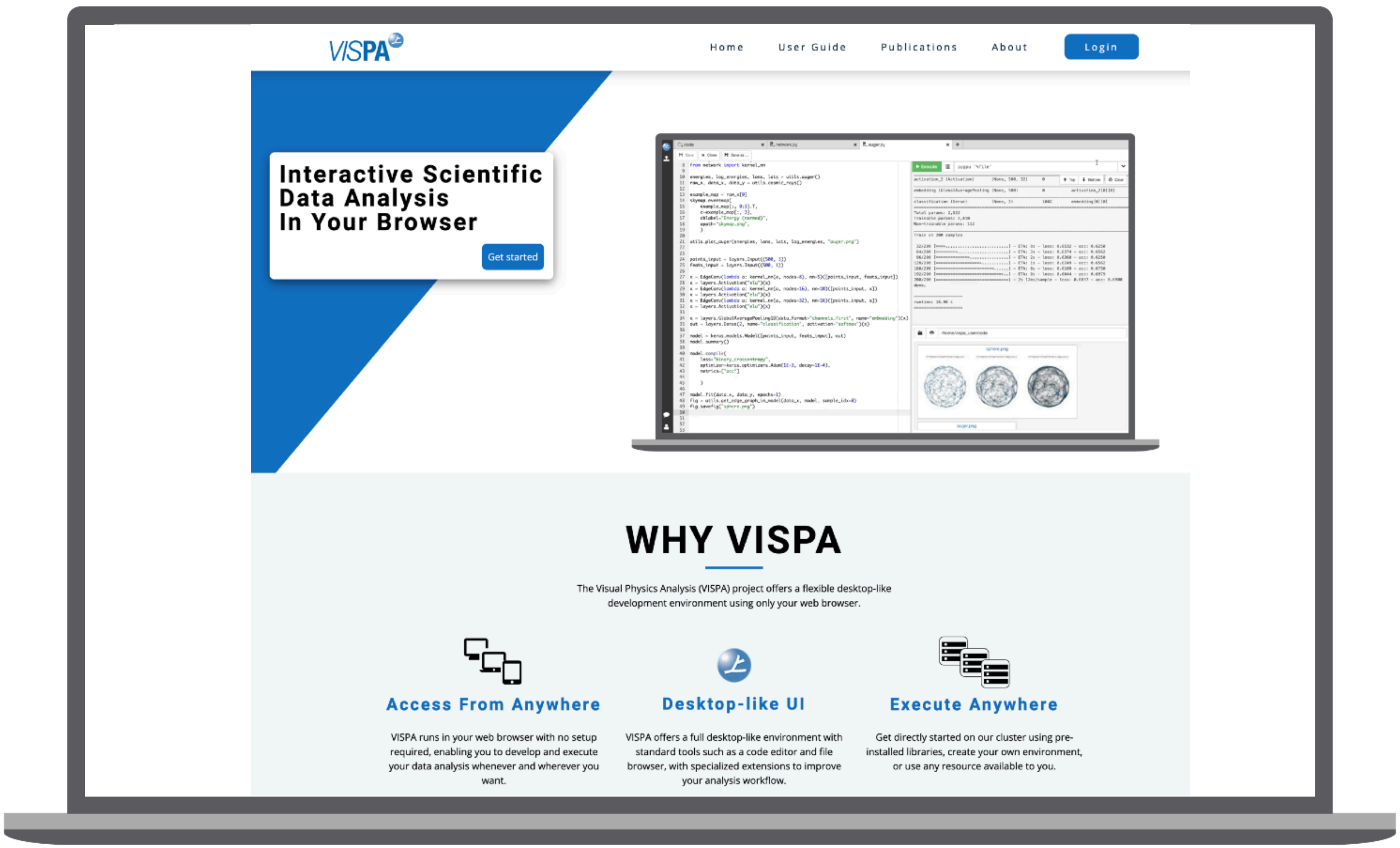
- All analyses run on local institute cluster
- Optimised for scientific data analysis and machine learning
- Hardware: see below
- Software:
  - System (Ubuntu OS) via ansible
  - Analysis via conda, shared over network



Total: CPU: 224T, RAM: 832GB, GPU: 17 (various)

Resource: VISPA Cluster (2/2)

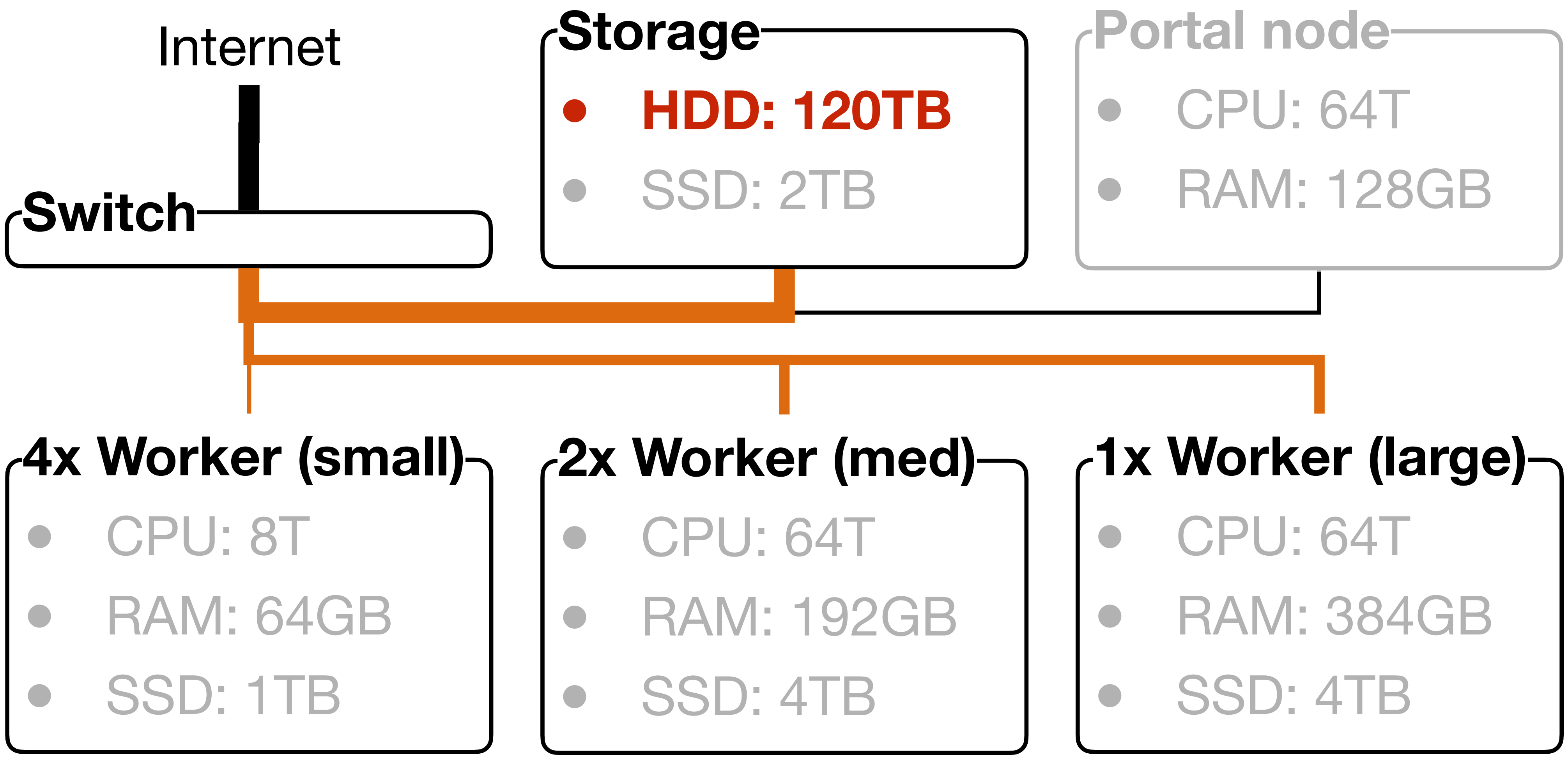
- User base:
  - 10 Researchers on daily basis (e.g. CMS experiment, Auger observatory)
  - Courses with up to 200 participants (e.g. Nuclear physics, ML in Physics)
  - Schools and workshop with up to 50 participants
- Front-end for data analysis in your web browser (link)



Live Demo

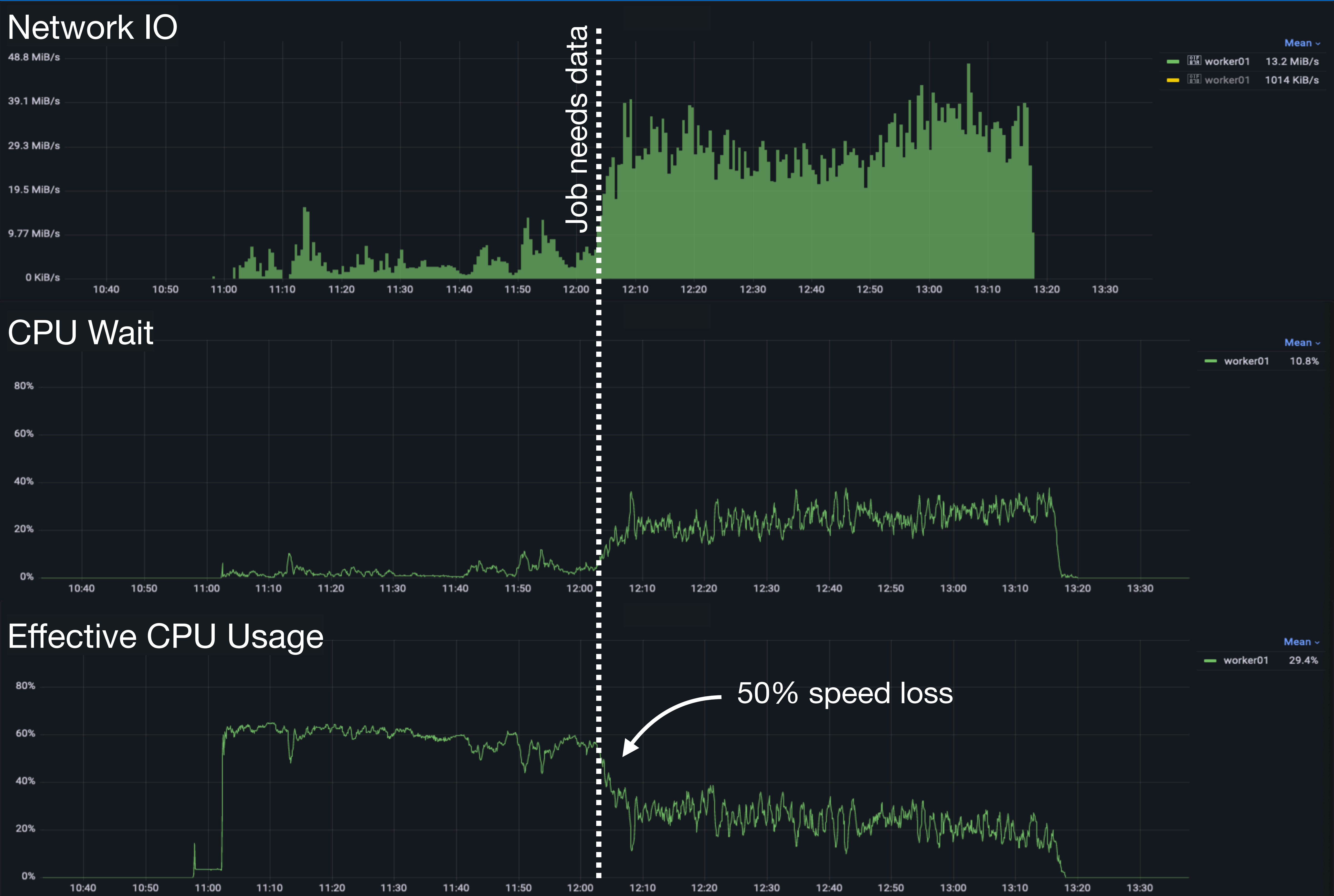
A New Bottleneck (1/2)

- Data is streamed from central network storage to worker nodes
- Map jobs are very time efficient and fast using vectorised processing
- Central network storage has two limitations:
  - **Read speed of HDDs**
  - **Limited network bandwidth**
- Streaming of data becomes critical bottleneck!



A New Bottleneck (2/2)

RWTH AACHEN UNIVERSITY
Dennis Noll - 17.10.22



Storage Layout

- Critical bottleneck is streaming of data to the processing elements
- Various storage types and locations available to solve the problem
- Two stage solution using central network storage and on-worker SSD

WLCG

O(PB)
O(100MB/s)

Many users
High latency

Storage Layout

RWTH AACHEN UNIVERSITY
Dennis Noll - 17.10.22

- Critical bottleneck is streaming of data to the processing elements
- Various storage types and locations available to solve the problem
- Two stage solution using central network storage and on-worker SSD

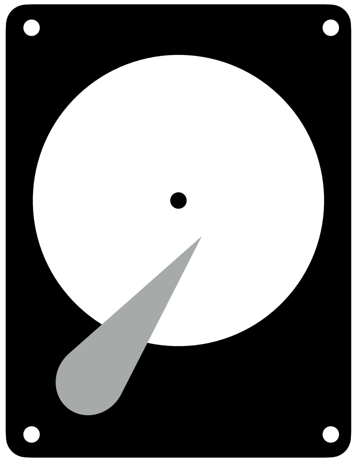
WLCG

WLCG
Worldwide LHC Computing Grid

O(PB)
O(100MB/s)

Many users
High latency

Storage



O(100TB)
O(100MB/s)

Stores experiment
data for our group

Slide 18

Storage Layout

- Critical bottleneck is streaming of data to the processing elements
- Various storage types and locations available to solve the problem
- Two stage solution using central network storage and on-worker SSD

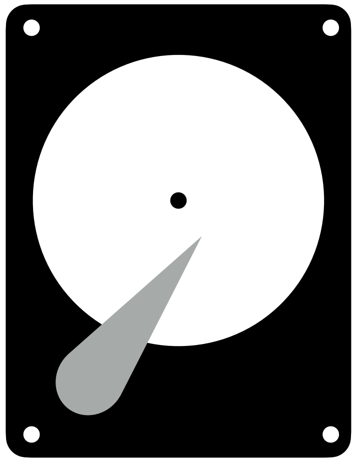
WLCG

WLCG
Worldwide LHC Computing Grid

O(PB)
O(100MB/s)

Many users
High latency

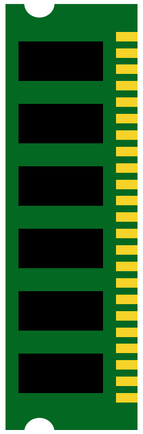
Storage

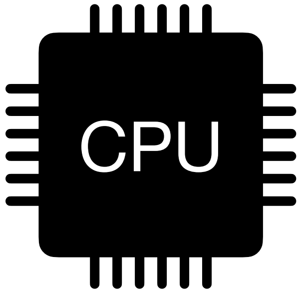


O(100TB)
O(100MB/s)

Stores experiment data for our group

Worker





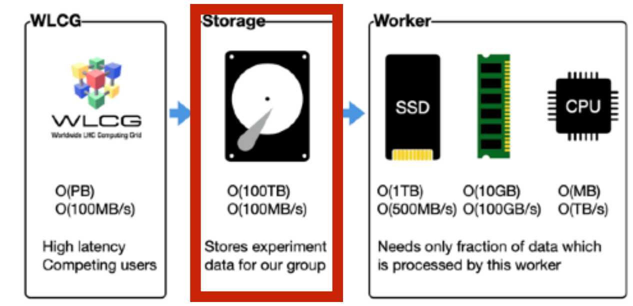
SSD	RAM	CPU
O(1TB)	O(10GB)	O(MB)
O(500MB/s)	O(100GB/s)	O(TB/s)

Needs only fraction of data which is processed by this worker

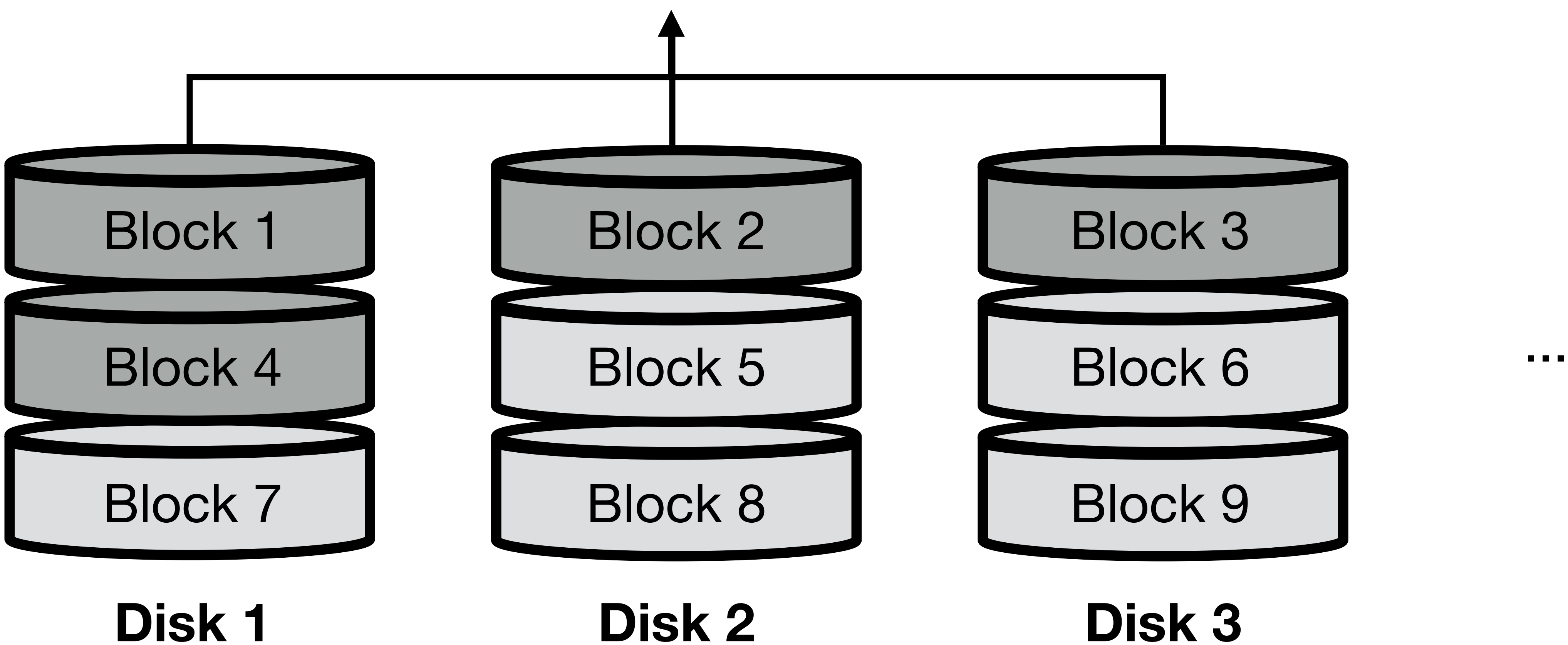
Slide 18

Slide 19-21

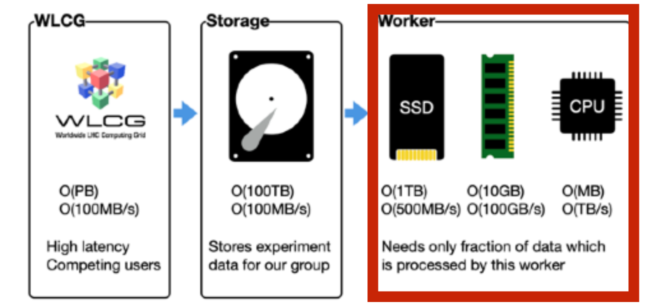
Network Attached Storage



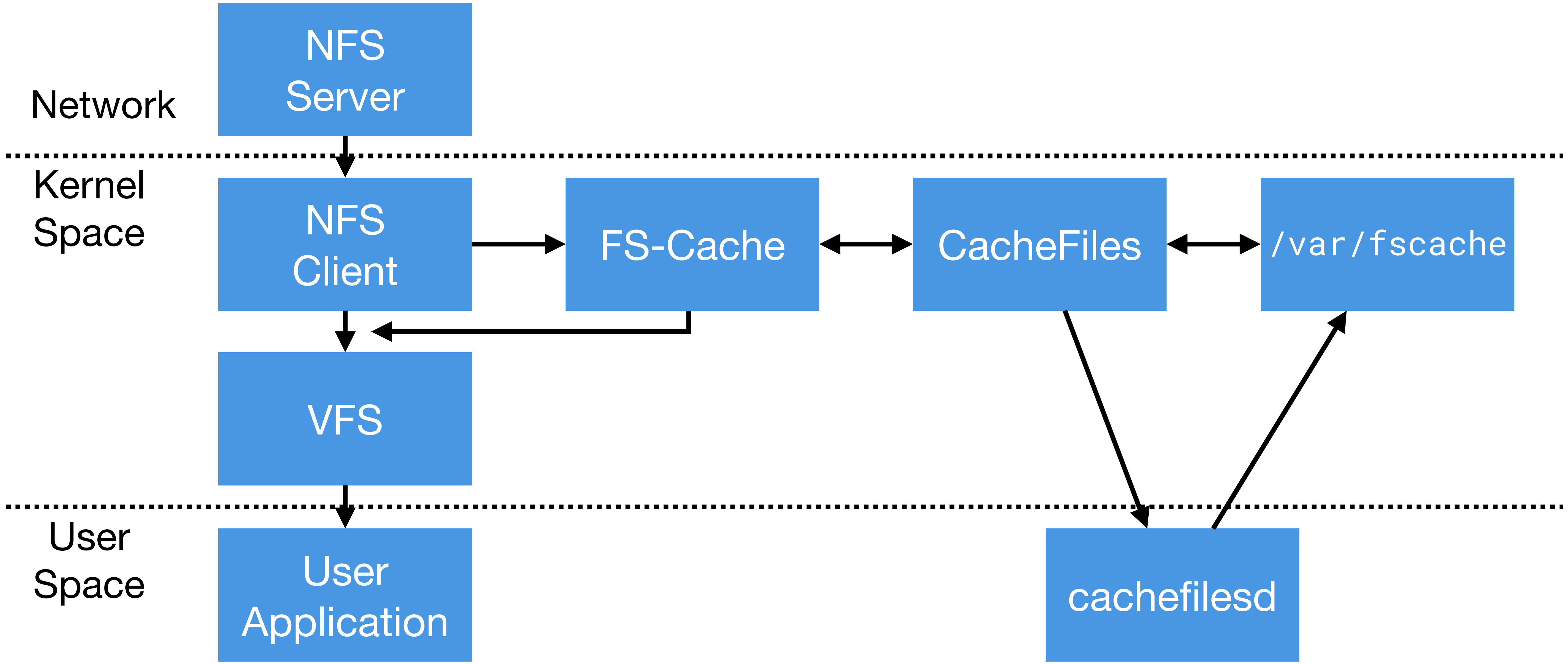
- Save experiment data on-site:
  - Easy and reliable access (no timeouts, credentials, ...)
  - Direct connection to worker nodes (low latency, 10GBit)
- Three storage tiers:
  - /home (2TB): User homes
  - /scratch (24TB): Mirrored experiment data
  - /store (96TB):
    - Un-mirrored experiment data
    - RAID0 striped across 6 x 16TB HDDs enables fast reading



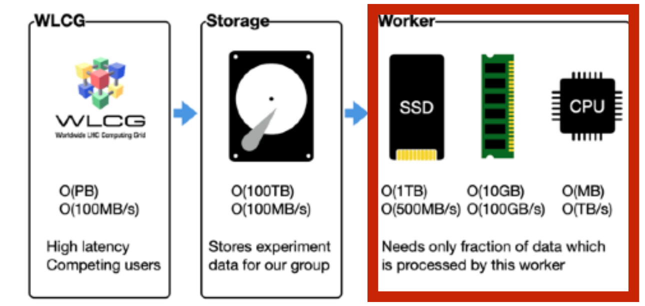
On-Worker Storage (FS-Cache)



- Using on-worker SSDs to minimise network traffic
- Used software implementation: FS-Cache & cachefilesd
  - Transparent caching system, available in Linux kernel
  - Granularity: Cache on block level (4kB)
  - Strategy: Least recently used (LRU)



On-Worker Storage (FS-Cache)



- Using on-worker SSDs to minimise network traffic
- Used software implementation: FS-Cache & cachefilesd
  - Transparent caching system, available in Linux kernel
  - Granularity: Cache on block level (4kB)
  - Strategy: Least recently used (LRU)

Network

Kernel Space

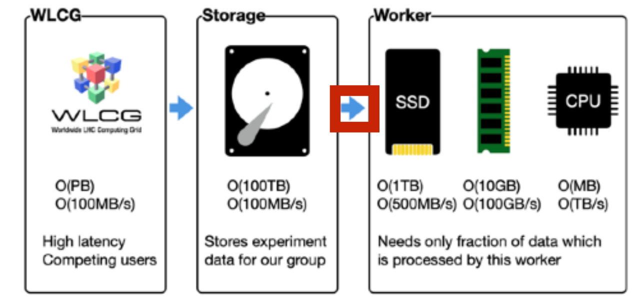
User Space

Application

Challenge: Cache trashing!

- Needed data gets evicted from cache
- Case 1: By somebody else
  - Everybody uses same files/chunks (where possible)
- Case 2: By yourself
  - Consistent assignment job ↔ worker via affine caching

Assignment Job ↔ Worker

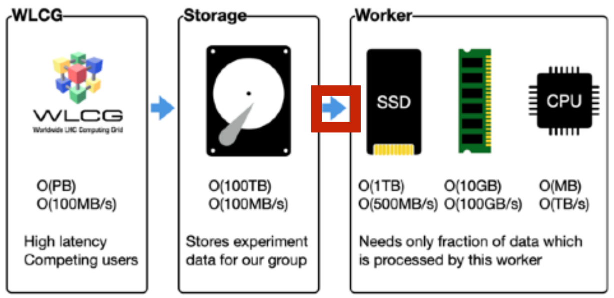


- Always process same data on same worker
- Use 64-dim embedding in hash space:
  - Embedding:

$$\texttt{"worker-01"} \xrightarrow{\text{sha512}} \texttt{10110100 11001001 ...}$$

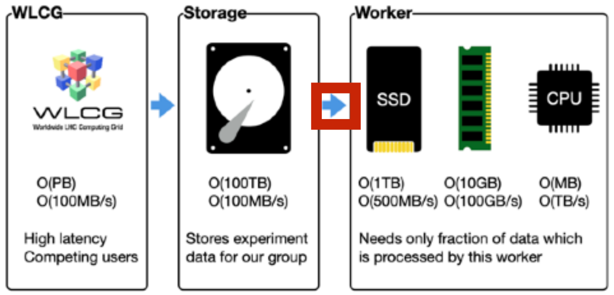
$$\texttt{10110100} \xrightarrow{\text{int}} x_1 = 180 \qquad \texttt{11001001} \xrightarrow{\text{int}} x_2 = 201$$

Assignment Job ↔ Worker

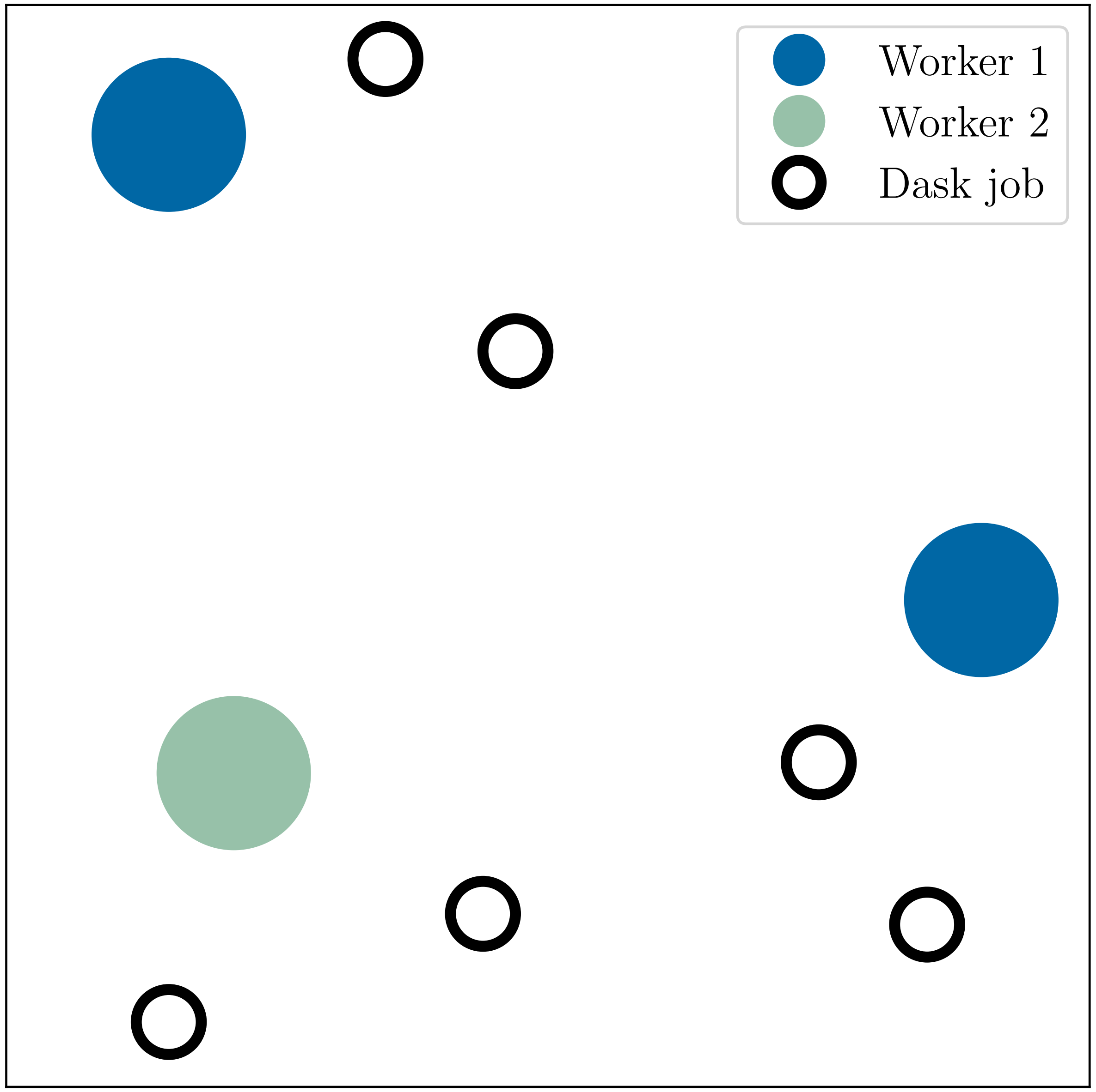


- Always process same data on same worker
- Use 64-dim embedding in hash space:
  - Embedding: embed("worker-01") = (180, 201, ...)

Assignment Job ↔ Worker



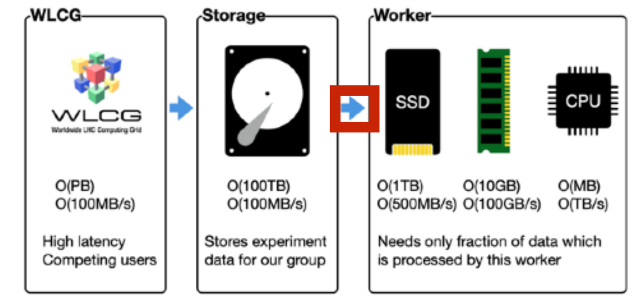
- Always process same data on same worker
- Use 64-dim embedding in hash space:
  - Embedding: embed("worker-01") = (180, 201, ...)
    - Worker: embed(worker_name + id)
    - Job/data: embed(filename + event range)



Logic similar to ceph (link)

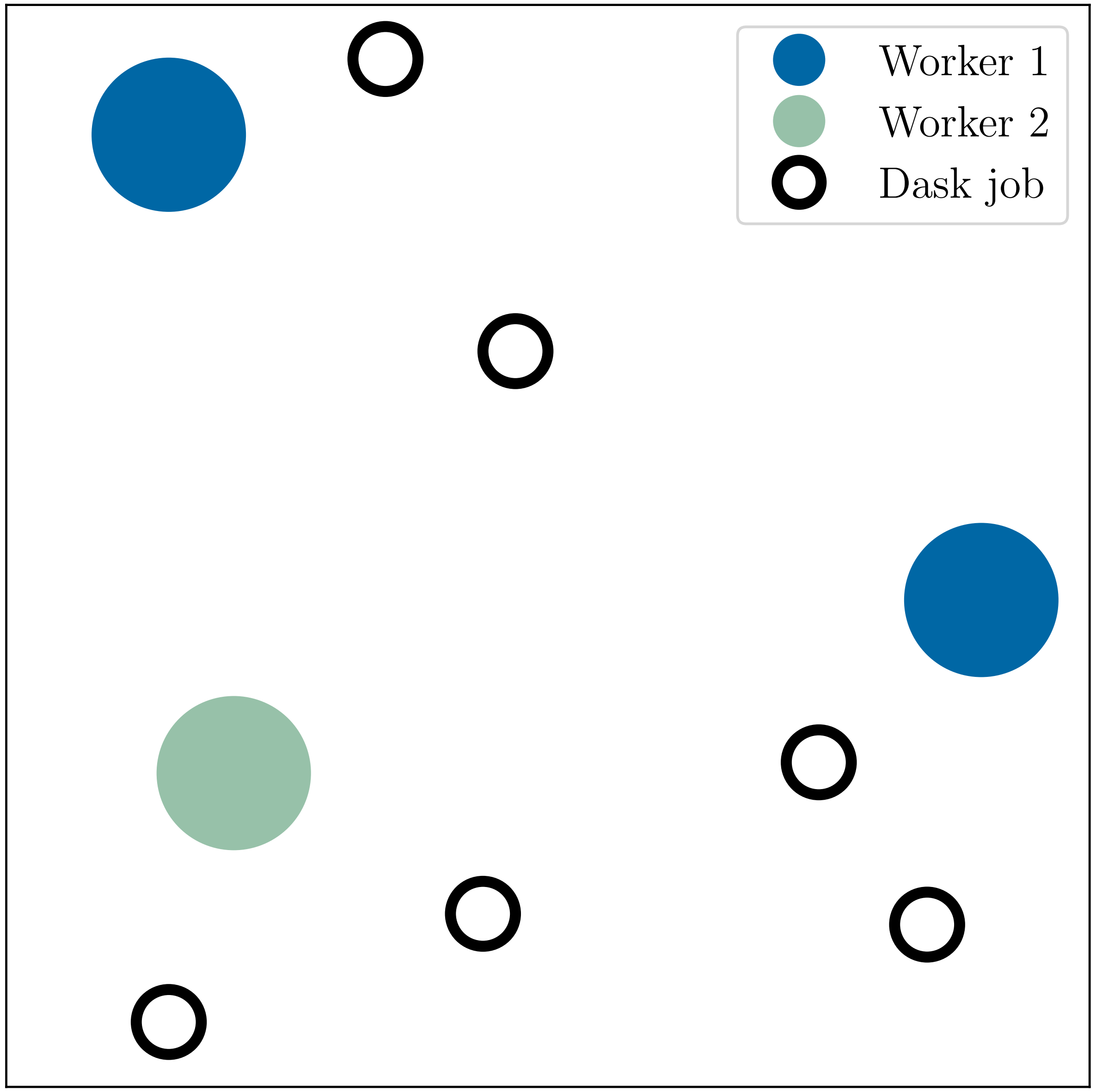
Assignment Job ↔ Worker

RWTH AACHEN UNIVERSITY
Dennis Noll - 17.10.22



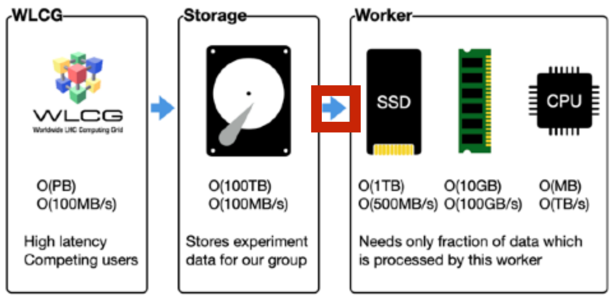
- Always process same data on same worker
- Use 64-dim embedding in hash space:
  - Embedding: embed("worker-01") = (180, 201, ...)
    - Worker: embed(worker_name + id)
    - Job/data: embed(filename + event range)

- Two tricks:
  - Worker have weight acc. to size
  - Place worker multiple times (id)



Logic similar to ceph (link)

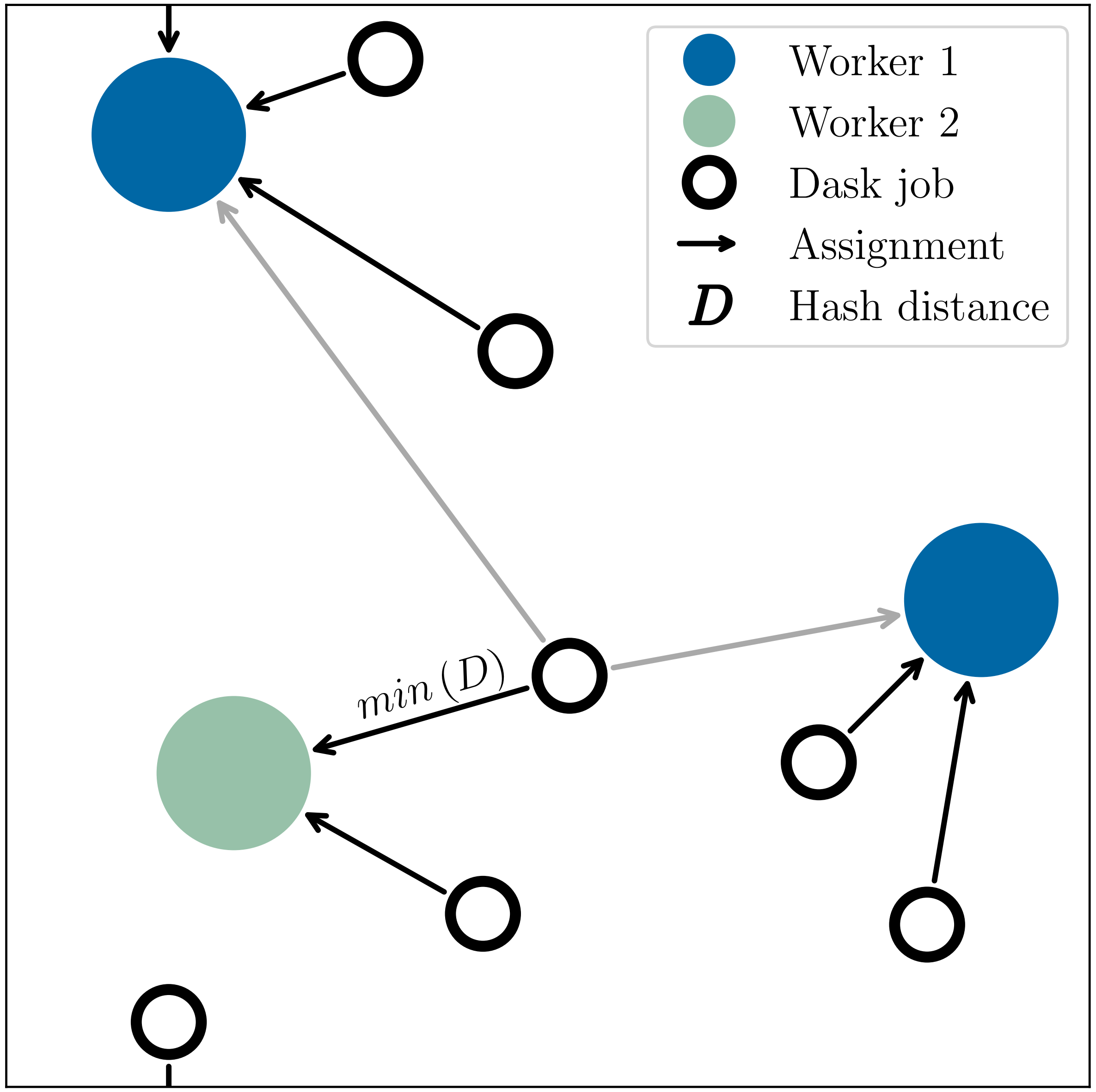
Assignment Job ↔ Worker



- Always process same data on same worker
- Use 64-dim embedding in hash space:
  - Embedding: embed("worker-01") = (180, 201, ...)
    - Worker: embed(worker_name + id)
    - Job/data: embed(filename + event range)

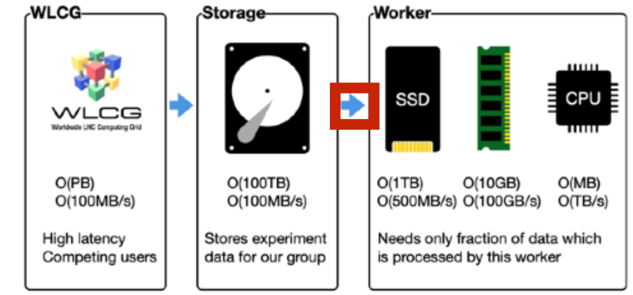
- Two tricks:
  - Worker have weight acc. to size
  - Place worker multiple times (id)

- D = Weighted euclidean distance
- Assignment via minimum D



Logic similar to ceph (link)

Assignment Job ↔ Worker

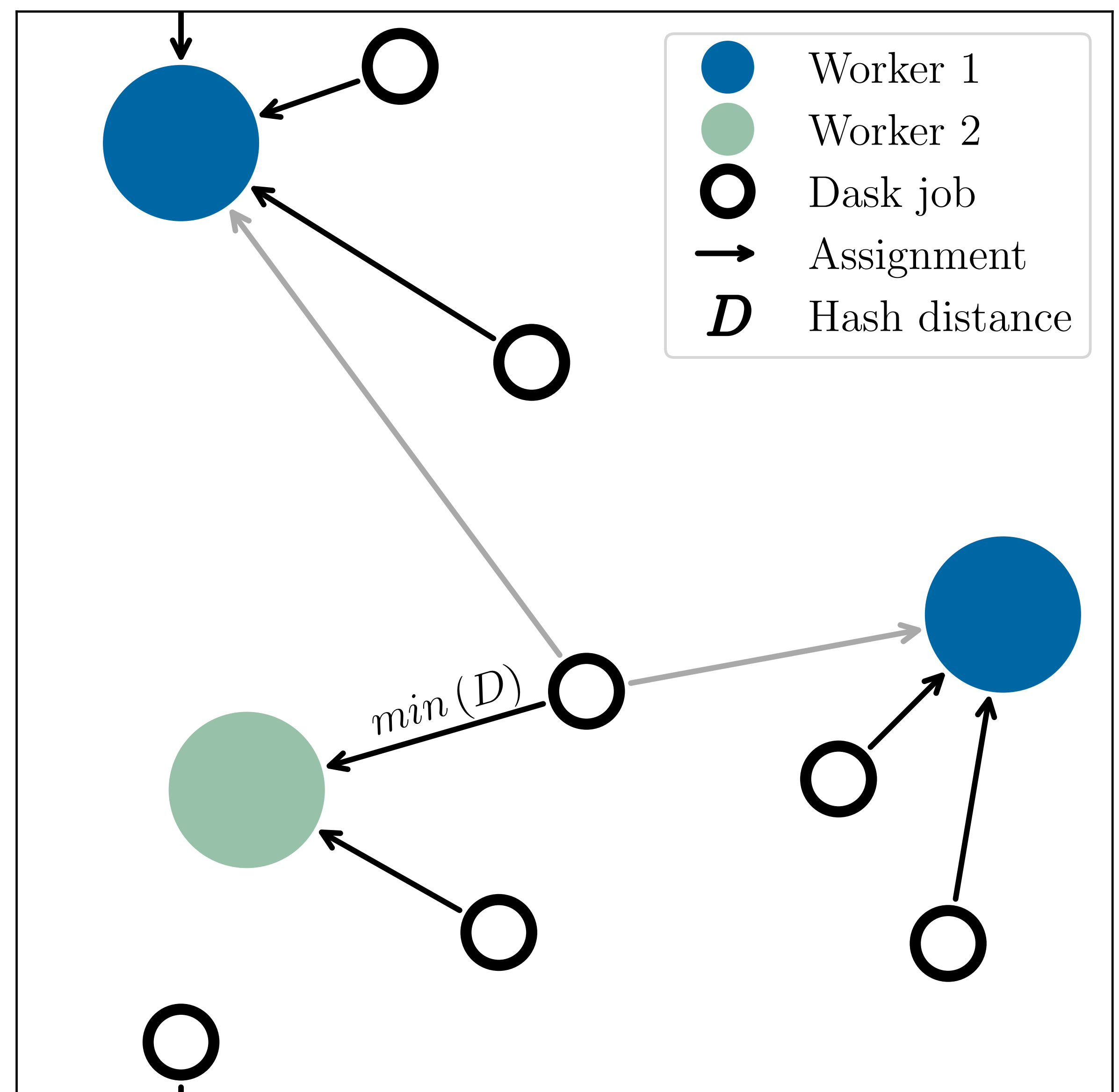


- Always process same data on same worker
- Use 64-dim embedding in hash space:
  - Embedding: embed("worker-01") = (180, 201, ...)
    - Worker: embed(worker_name + id)
    - Job/data: embed(filename + event range)

- Two tricks:
  - Worker have weight acc. to size
  - Place worker multiple times (id)

- D = Weighted euclidean distance
- Assignment via minimum D

- Results in:
  - Affine caching
  - Graceful on failures



Logic similar to ceph (link)

Many Alternatives! (& why we don't use them)

XCache

XRootD

- No POSIX access
- Not easy with many users

Distributed File Systems

ceph

- Does not handle changing files so well

XTREEMFS

- No cache tiering

GlusterFS

- Cache tiering removed

lustre File System

- No graceful failure of parts of storage

MooseFS

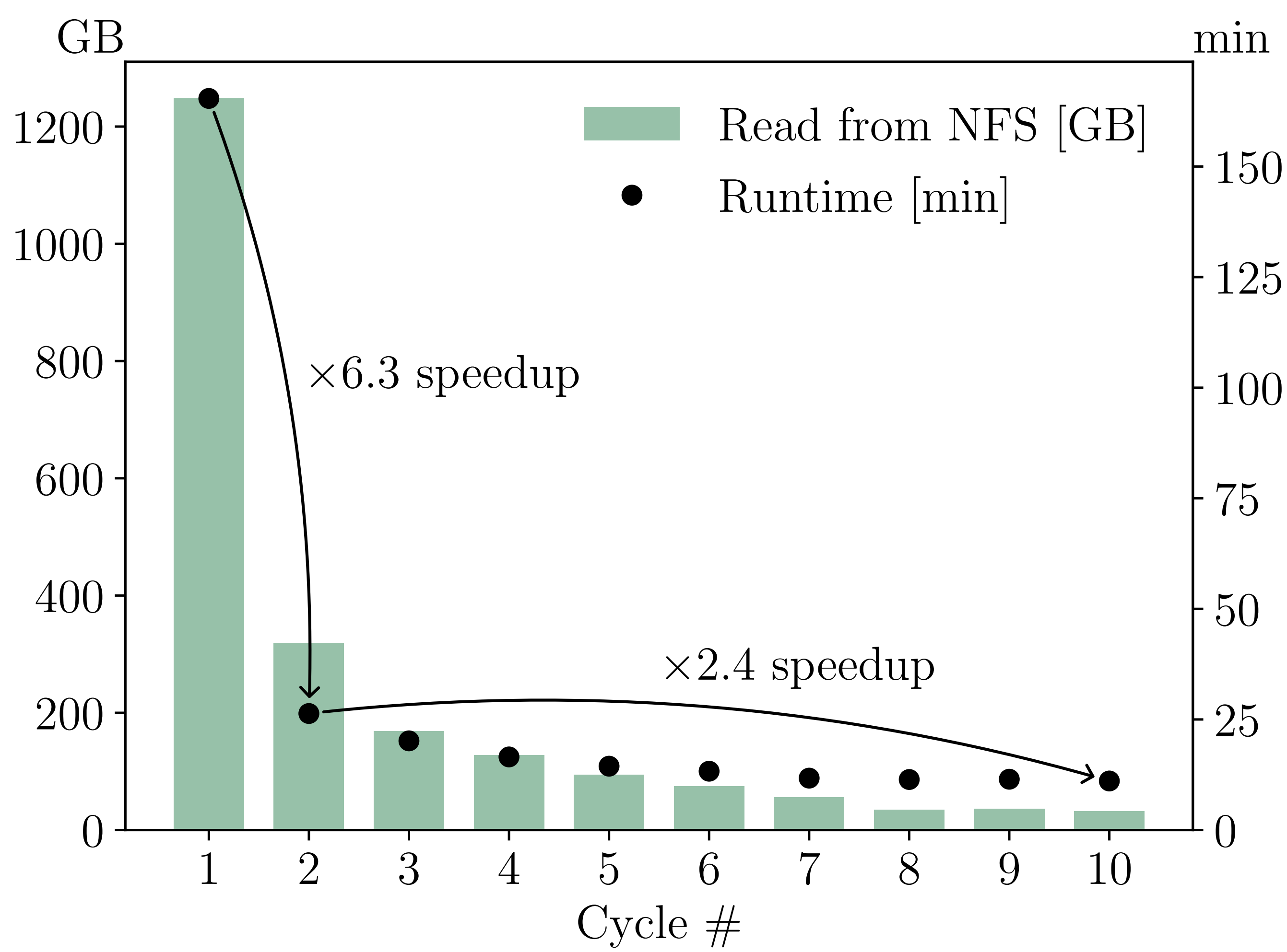
- Caching only based on modification time

Seaweed

- Seems promising but still very new
- Documentation not complete yet

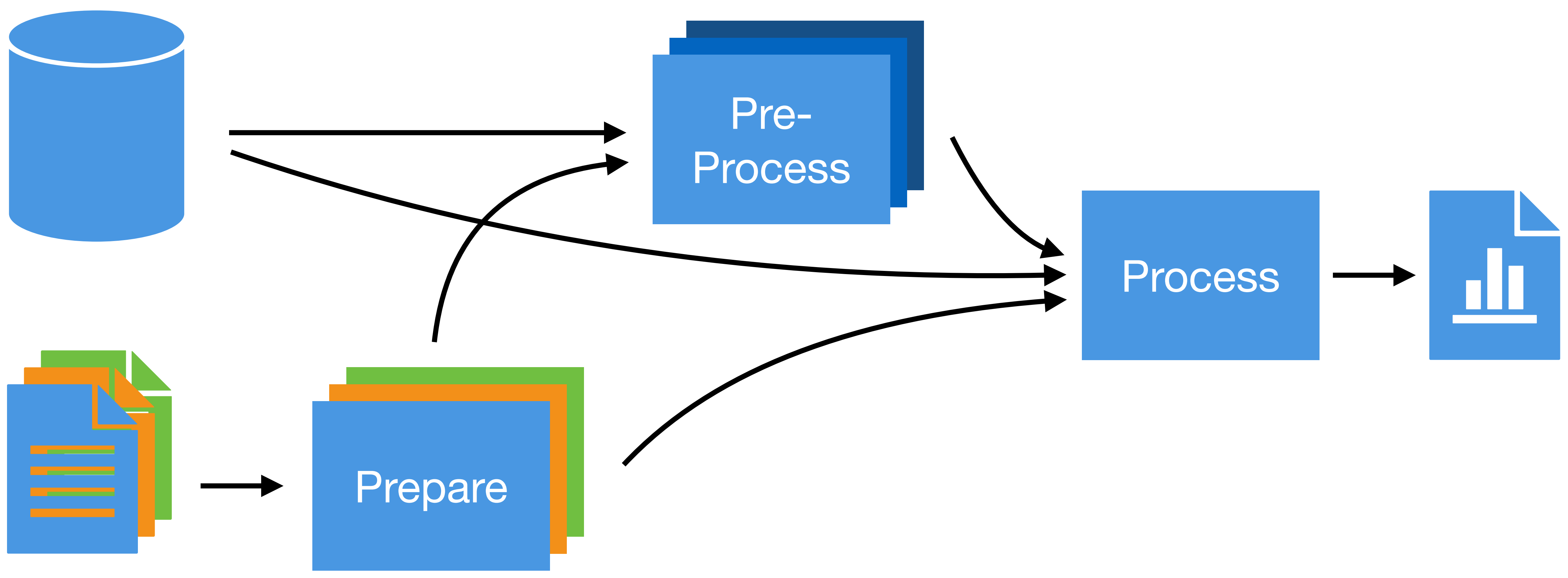
Benchmark: Definition and Results

- Read-only task run 10 times (cycles) with 220 workers
- Data:
  - Higgs pair production analysis (1440GB, 10^9 events, 120 columns)
  - Using read-optimised compression algorithm (Z-std, 10)
- Results:
  - **Gradual performance: work stealing**
  - **Runtime lower bound: CPU → IO Bottleneck solved**



Impact on Analysis

- Particular speedup for read heavy pre-processors (e.g. PU counting)
- Enables different analysis run-modes for full analysis (e.g. 2016, 3TB):
  - **Explorative** (w/o correction & systematic shifts, only simulations): **20min**
  - **Quick** (w/o computation heavy systematic variations (JEC)): **6h**
  - **Full** publication-ready analysis run: **20h**



Summary

- Fast O(TB) Physics Analysis on Small Institute Cluster
- Columnar processing via NumPy, Awkward
- Job distribution via map & reduce (Dask)
- Solved IO bottleneck:
  - Optimized storage distribution
  - Affine caching concept
- Analysis runtimes:
  - Explorative: 20 min
  - Full: 20h