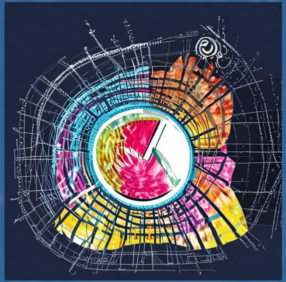
CERN

CHEP 2024



CERN

ROOT RNTuple & EOS

A holistic approach to data analysis

Chapter 1: **ROOT RNTuple - Next Generation Event Data I/O for HENP**
Chapter 2: **Exploring Client-Server Scalability with RNTuple & EOS**

Presenters: Jakob Blomer & Andreas Joachim Peters

EOS
OpenStorage

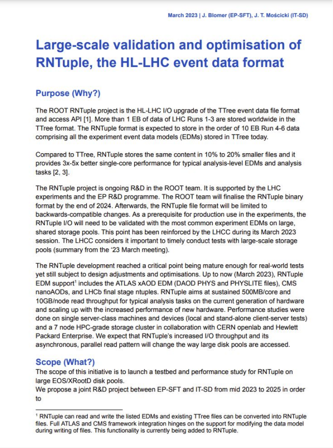
ROOT
Data Analysis Framework

https://root.cern

Part of the work presented is formalized in a two years joint project between the EP-SFT and IT-ST group at CERN:

Large-scale validation and optimisation of RNTuple, the HL-LHC event data format

[PSO]



March 2023 | J. Blomer (EP-SFT), J. T. Mościcki (IT-SD)

Large-scale validation and optimisation of RNTuple, the HL-LHC event data format

Purpose (Why?)

The ROOT RNTuple project is the HL-LHC I/O upgrade of the TTree event data file format and access API [1]. More than 1 EB of data of LHC Runs 1-3 are stored worldwide in the TTree format. The RNTuple format is expected to store in the order of 10 EB Run 4-6 data comprising all the experiment event data models (EDMs) stored in TTree today.

Compared to TTree, RNTuple stores the same content in 10% to 20% smaller files and it provides 3x-5x better single-core performance for typical analysis-level EDMs and analysis tasks [2, 3].

The RNTuple project is ongoing R&D in the ROOT team. It is supported by the LHC experiments and the EP R&D programme. The ROOT team will finalise the RNTuple binary format by the end of 2024. Afterwards, the RNTuple file format will be limited to backwards-compatible changes. As a prerequisite for production use in the experiments, the RNTuple I/O will need to be validated with the most common experiment EDMs on large, shared storage pools. This point has been reinforced by the LHCC during its March 2023 session. The LHCC considers it important to timely conduct tests with large-scale storage pools (summary from the '23 March meeting).

The RNTuple development reached a critical point being mature enough for real-world tests yet still subject to design adjustments and optimisations. Up to now (March 2023), RNTuple EDM support[1] includes the ATLAS xAOD EDM (DAOD PHYS and PHYSLITE files), CMS nanoAODs, and LHCb final stage ntuples. RNTuple aims at sustained 500MB/core and 10GB/node read throughput for typical analysis tasks on the current generation of hardware and scaling up with the increased performance of new hardware. Performance studies were done on single server-class machines and devices (local and stand-alone client-server tests) and a 7 node HPC-grade storage cluster in collaboration with CERN openlab and Hewlett Packard Enterprise. We expect that RNTuple's increased I/O throughput and its asynchronous, parallel read pattern will change the way large disk pools are accessed.

Scope (What?)

The scope of this initiative is to launch a testbed and performance study for RNTuple on large EOS/XRootD disk pools.
We propose a joint R&D project between EP-SFT and IT-SD from mid 2023 to 2025 in order to

[1] RNTuple can read and write the listed EDMs and existing TTree files can be converted into RNTuple files. Full ATLAS and CMS framework integration hinges on the support for modifying the data model during writing of files. This functionality is currently being added to RNTuple.

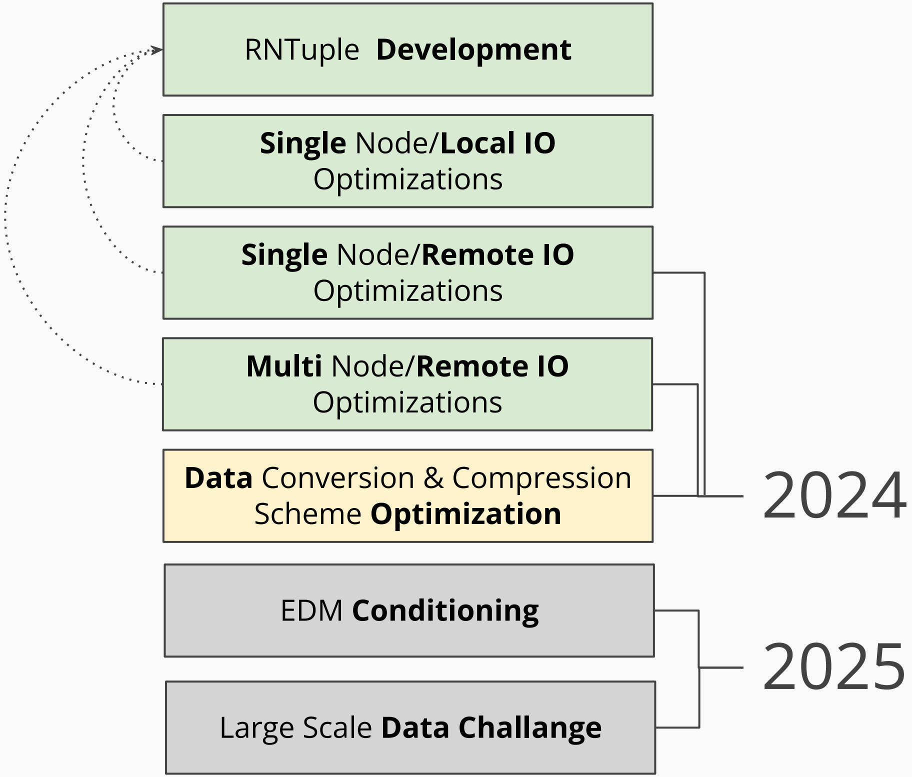
This activity is part of the

RCS-IT

Research and Computing Sector & CERN IT Engagement

Scope & Approach

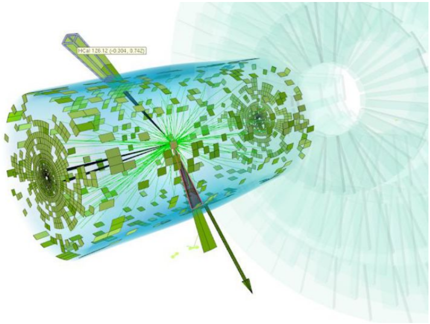
- **Local IO optimizations**
  - End-user analysis
  - Data conversion tools
  - Parallel Writing
- **Remote IO optimizations**
  - Scale-out for Analysis
  - Facility Usage
- **Data Format optimizations**
  - Compression schemes
  - IO sizes
- **Final Validation**
  - Data Challenge



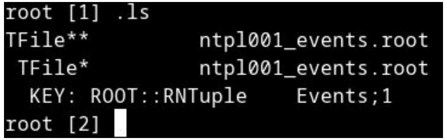
Chapter 1

ROOT RNTuple

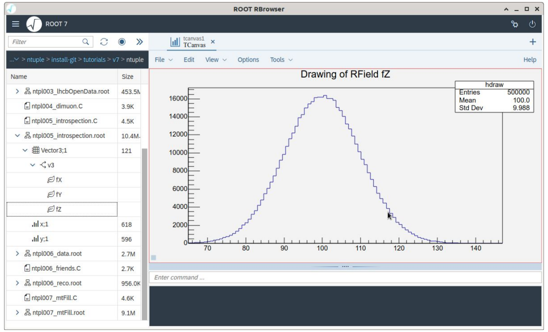
RNTuple: redesigned columnar I/O for HENP



>2EB (now) → >10EB (end of HL-LHC)
~½ of the WLCG budget on storage



ROOT File with RNTuple



RBrowser showing RNTuple data

- Building on 25+ years of TTree experience, modern & efficient implementation:
  - **Smaller files** (typically 10% - 50%), **higher throughput** (often by factors)
  - More **robust**: binary format specification, modern API, fully checksummed
  - Efficient support of **modern storage systems**: NVMe, object stores, async & parallel I/O
  - Forward-looking limits: designed for TB-sized events and PB-sized files
- Feature-rich: works with **complex experiment EDMs** and with experiment frameworks
- Supported at HL-LHC timescale (2040+)

HENP I/O Challenge: Rich Event Data Models

Type Class	Types	EDM Coverage	RNTuple Status
PoD	`bool, char, std::byte, (u)int[8,16,32,64]_t, float, double`		Available
Records	Manually built structs of PoDs	Flat n-tuple	
(Nested) vectors	`std::vector, RVec, std::array,` C-style fixed-size arrays	Reduced AOD	Available
String	`std::string`		Available
User-defined classes	Non-cyclic classes with dictionaries		Available
User-defined enums	Scoped / unscoped enums with dictionaries	Full AOD / ESD / RECO	Available
User-defined collections	Non-associative collection proxy		Available
stdlib types	`std::pair, std::tuple, std::bitset, std::(unordered_)(multi)set, std::(unordered_)(multi)map`		Available
Alternating types	`std::variant, std::unique_ptr, std::optional`		Available
Streamer I/O	All ROOT streamable objects (stored as byte array)		Available
Low-precision floating points	`Double32_t, f16`	*Optimization benefitting all EDMs*	Available
	Custom precision / range (bfloat16, TensorFloat-32, other AI formats)		Available

HENP I/O Challenge: Rich Event Data Models

Type Class	Types	EDM Coverage			RNTuple Status
PoD	`bool, char, std::byte, (u)int[8,16,32,64]_t, float, double`	Flat n-tuple	Reduced AOD	Full AOD / ESD / RECO	Available
Records	Manually built structs of PoDs				
(Nested) vectors	`std::vector, RVec, std::array,` C-style fixed-size arrays				Available
String	`std::string`				Available
User-defined classes	Non-cyclic classes with dictionaries				Available
User-defined enums	Scoped / unscoped enums with dictionaries				Available
User-defined collections	Non-associative collection proxy				Available
stdlib types	`std::pair, std::tuple, std::bitset, std::(unordered_)(multi)set, std::(unordered_)(multi)map`				Available
Alternating types	`std::variant, std::unique_ptr, std::optional`				Available
Streamer I/O	All ROOT streamable objects (stored as byte array)				Available
Low-precision floating points	`Double32_t, f16`	*Optimization benefitting all EDMs*			Available
	Custom precision / range (bfloat16, TensorFloat-32, other AI formats)				Available

Limit of HDF5 and Big Data formats (e.g., Parquet)

HENP I/O has unique requirements and challenges

1. Natural HENP data layout is
   jagged arrays of complex types with columnar access pattern
   - HDF5 does not fit well due to its inherent tensor layout
   - Otherwise only found in Big Data (but with limited type support)

2. HENP data organization: global federation of file sets
   - Requires **XRootD** and **HTTP** remote data access
   - Extra functionality to build data sets from files:
     ***fast* merging, chains,** 👉 joins

3. Integration in the HENP software landscape
   - **Rich type system** of experiment central EDMs with **10k+ columns**
   - Multi-threaded reading and writing under tight memory constraints
   - Schema extension during writing
   - Availability in the Python & C++ analysis ecosystem, e.g. in ROOT **RDataFrame**

4. >10 EB of data to be stored over decades
   - Requires **excellent compression** (lossy and lossless)
   - Data custodianship over time:
     backward & forward compatibility, schema evolution, bit-level checksumming

Event data model, very simplified

```
struct Hit {
   float x, y, z;
};

struct Particle {
   float fE;
   std::vector<Hit> fHits;
};

struct Event {
   int fEventNo;
   std::vector<Particle> fParticles;
};
```

In reality:
>1000 data classes
>10k properties
billions of events
decades of retention

Due to ROOT's C++ interpreter:
classes *are* the schema

RNTuple in Practice

- For maximum optimization opportunities, RNTuple introduces a **new on-disk format** and a **new API**

- At the same time, RNTuple is **smoothly integrated** with the established ROOT/HENP ecosystem.
  - RNTuple data stored in **ROOT files**
  - Consistent tooling
    - **RBrowser** support
    - **TFileMerger** & **hadd** support
    - **Disk-to-disk converter** TTree ➔ RNTuple [1]
  - RNTuple **adopts TTree's I/O customization and schema evolution** system
  - For **RDataFrame** code: no change required
  - Based on the specification: 3rd party readers available, e.g. for 👉Julia

```
root [1] .ls
TFile**         /data/gg_data.root
 TFile*         /data/gg_data.root
  KEY: TTree    mini;55 mini [current cycle]
  KEY: TTree    mini;54 mini [backup cycle]
  KEY: ROOT::RNTuple    mini_imported;1 object title
root [2]
```

A TTree and an RNTuple in the same ROOT file. In this example, the RNTuple data has been converted from the tree using the `RNTupleImporter`.

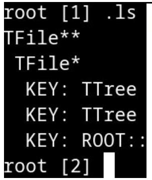
- For frameworks and power users, RNTuple provides a **modern API for (multi-threaded) writing and reading**
  - Follows C++ core guidelines
  - e.g., smart pointers, runtime errors signaled by exceptions
  - Reviewed by HEP-CCE

Argonne NATIONAL LABORATORY   Brookhaven National Laboratory   Fermilab   BERKELEY LAB

RNTuple in Practice

- For maximum optimization opportunities, RNTuple introduces a **new on-disk format** and a **new API**

- At the same time, RNTuple is **smoothly integrated** with the established ROOT/HENP ecosystem.
  - RNTuple data stored in **ROOT files**
  - Consistent tooling
    - **RBrowser** support
    - **TFileMerger** & **hadd** support
    - **Disk-to-disk converter** TTree ➜ RNTuple [1]
  - RNTuple **adopts TTree's I/O customization and schema evolution** system
  - For **RDataFrame** code: no change required
  - Based on the specification: 3rd party readers available, e.g. for 👉Julia

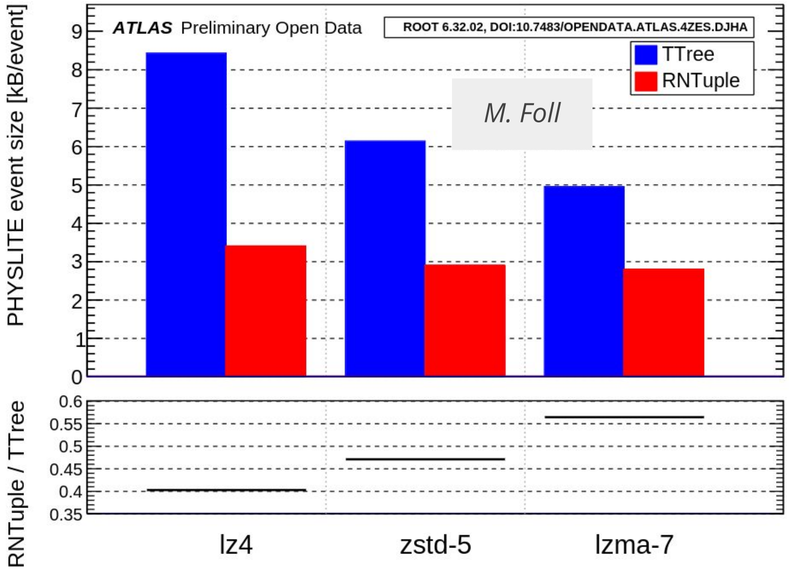


A TTree and an R
the RNTuple da

- For frameworks and power users, RNTuple provides a **modern API for (multi-thre**
  - Follows C++ core guidelines
  - e.g., smart pointers, runtime errors signaled by exceptions
  - Reviewed by HEP-CCE

Argonne NATIONAL LABORATORY · Brookhaven National Laboratory · Fermilab · BERKELEY LAB

RNTuple Performance: Data Volume



Example: ATLAS DAOD

RNTuple in ATLAS 👉 [1] 👉 [2] 👉 [3]

Note that due to data preconditioning in RNTuple, the relative difference between compression algorithms fades.

Contributors to space savings

- More compact on-disk representation of collections and booleans (trigger bits)
- Same page merging
- Type-based data encoding optimized for better compression ratio

Also: new default settings,
e.g. **zstd** compression algorithm

More performance studies

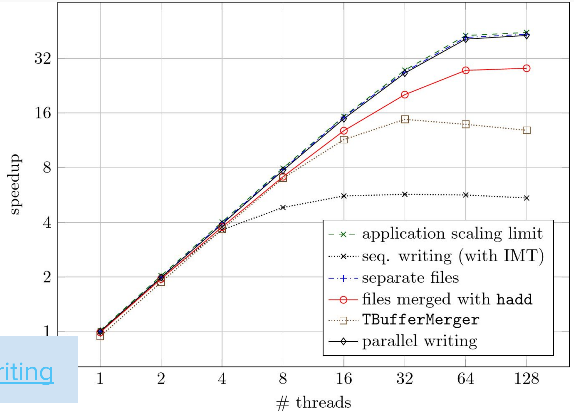
- 👉 CMS
- 👉 LHCb
- 👉 Comparison with HDF5 & Parquet (ACAT 21)

RNTuple Performance: Throughput Examples

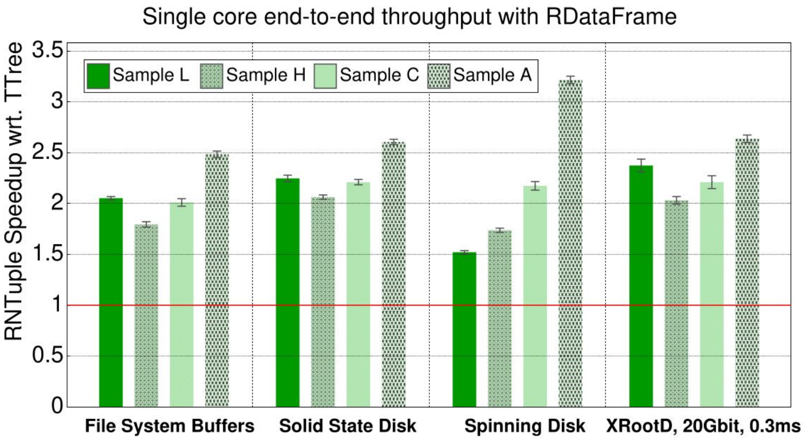
Higher single-core RDataFrame read throughput across various final-stage ntuple types and data access modes.

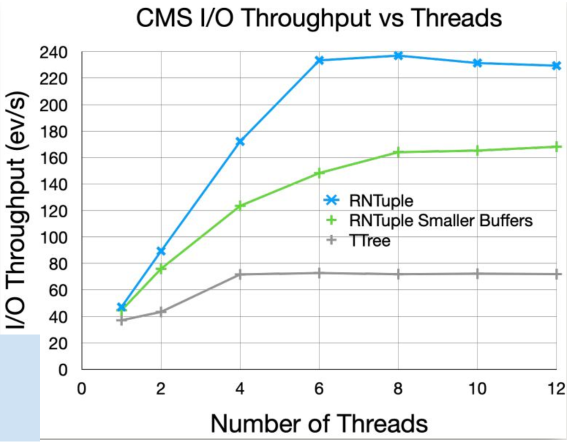
Contributors to higher throughput

- Asynchronous prefetching
- Multi-stream disk access through `io_uring`
- Code optimization
- New on-disk layout allows for higher degree of explicit and implicit parallelization
- New RDataFrame I/O scheduler



👉 Parallel & Direct I/O writing





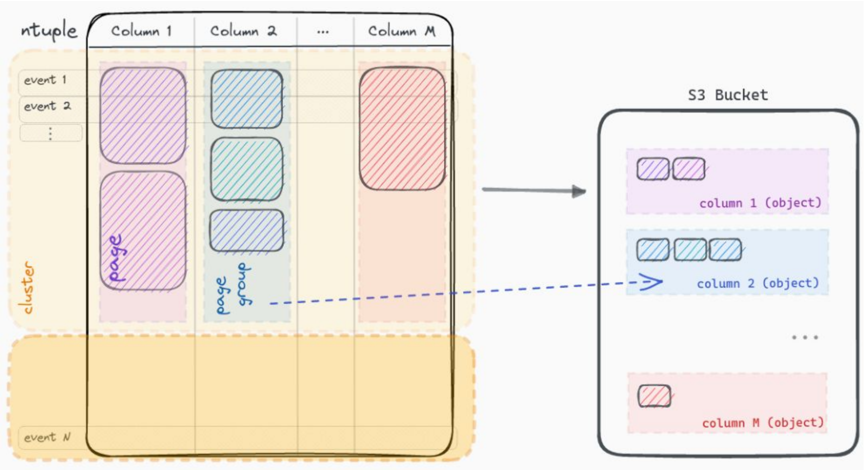
Better IMT scalability
👉 CMS

Beyond Files: Support for Object Stores

- Object store (S3) is the primary storage technology in the cloud
- A storage option for HPC, too (e.g., DAOS on Aurora)
- RNTuple is built such that its data blocks can be directly mapped to an object store.
- **Pre-production implementation for DAOS**
  - Prototype implementation for S3
- **RNTuple design gives access to native performance of object stores**

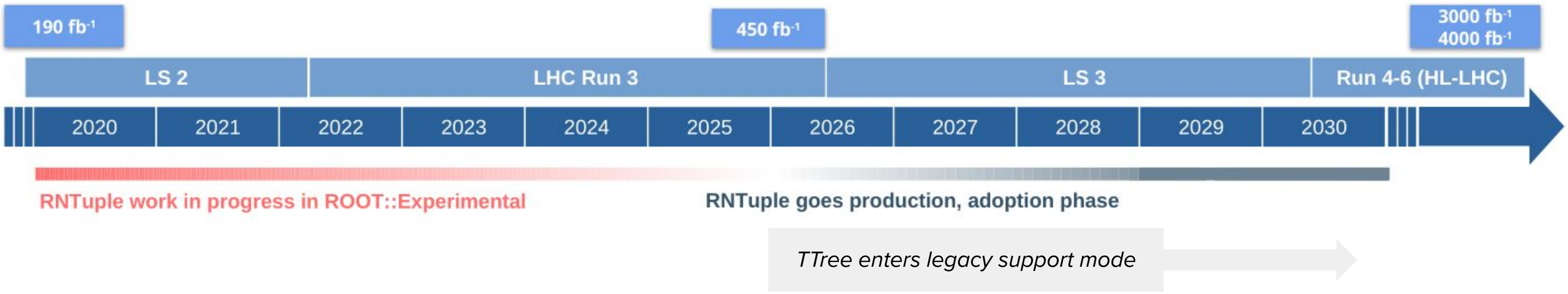
Native RNTuple object store support reaches 2GB/s/client.
The file system emulation layer peaks at 250MB/s/client.

👉 RNTuple on DAOS



Example of mapping onto S3 objects

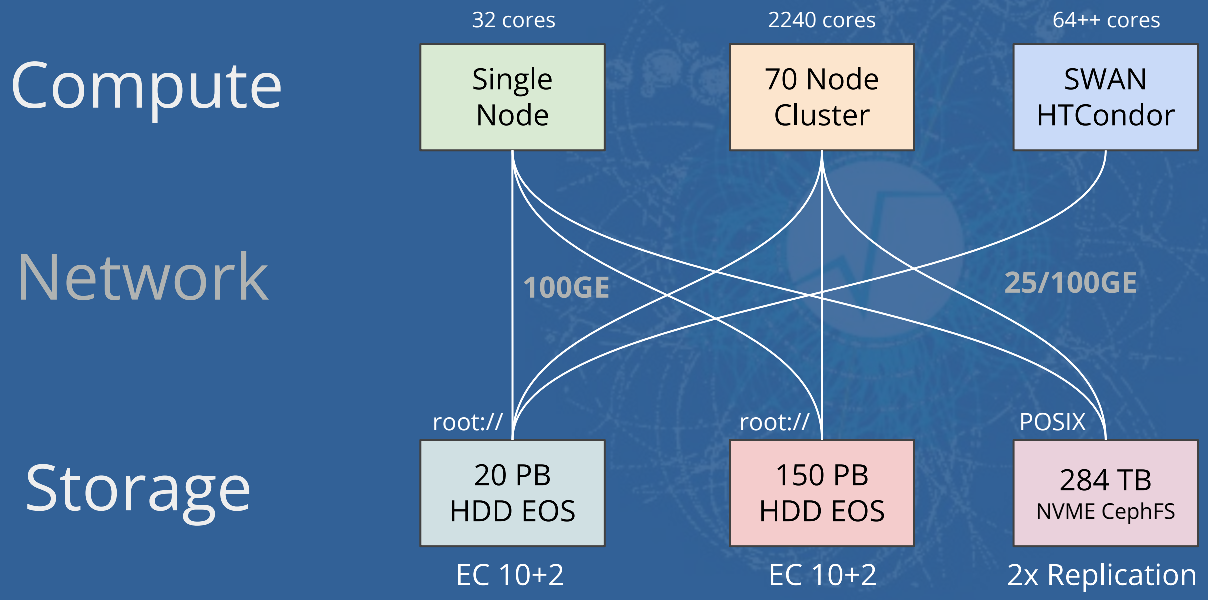
RNTuple is a HENP I/O system for the HL-LHC era

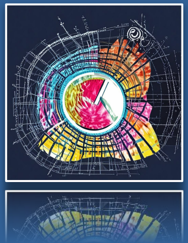


- **RNTuple is transitioning into production**
  - Major milestone after 6 years of R&D
    - Final first version of on-disk format for ROOT 6.34 (November 2024)
    - Future ROOT versions will be backwards-compatible
  - First production API: ROOT 6.36 (H1 / 2025)
    - Will incorporate the HEP-CCE API review suggestions
- **Start of first exploitation phase in 2025**
- **RNTuple provides a solid basis for future I/O R&D**

Chapter 2

RNTuple with Remote Storage





See also **Next-Gen Storage Infrastructure for ALICE** *https://indico.cern.ch/event/1338689/contributions/6010773/*

Large Scale Validation - Introduction

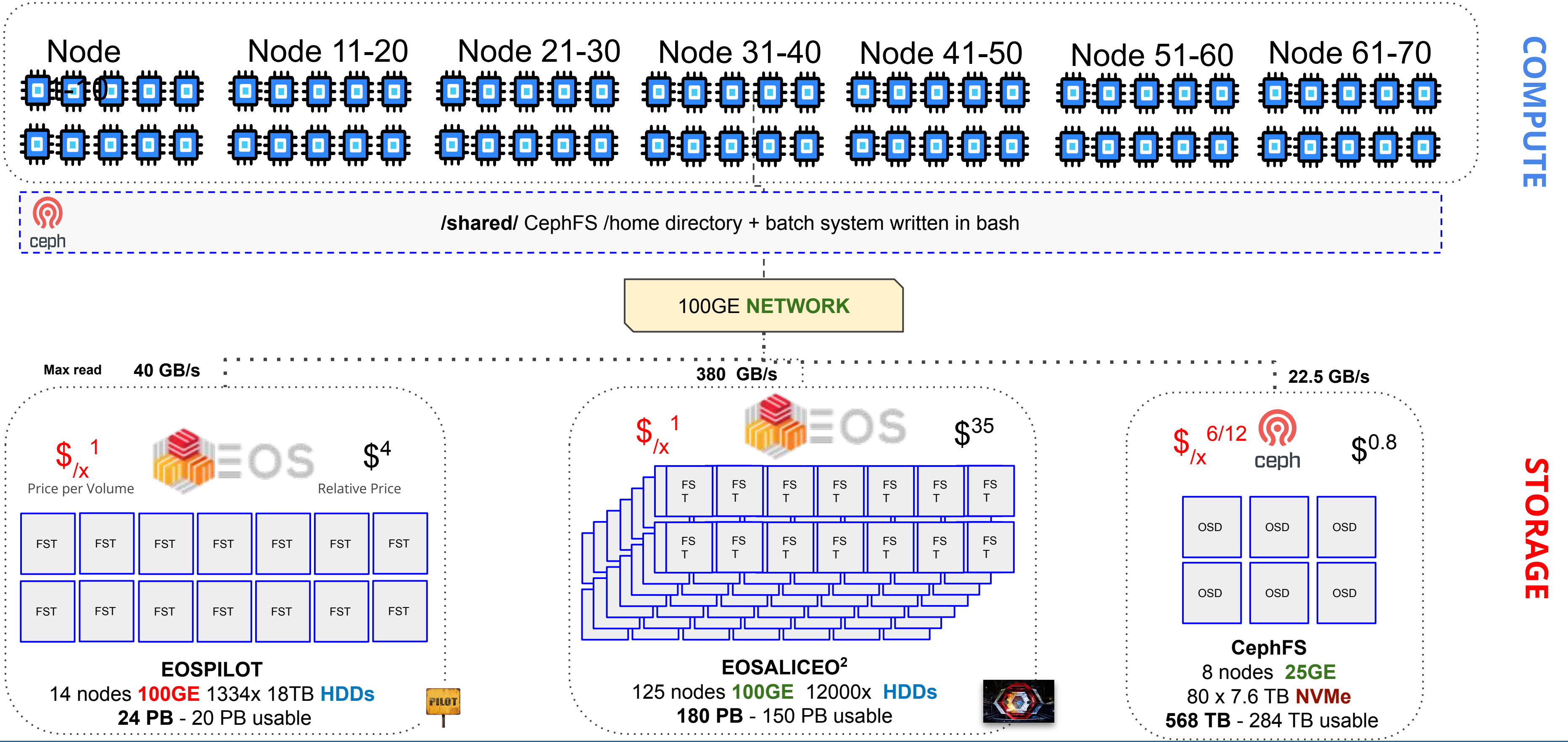
- We set up a bare-metal computing environment, EO²C, with 70 nodes on 100GE for a **large-scale validation** study and benchmarked it using three different storage backends.

- The performance studies used an **Analysis Grand Challenge** example using **RDataFrame**

    - In all measurements, the dataset is **uncached** both in the back-end and, where applicable, in the client cache.

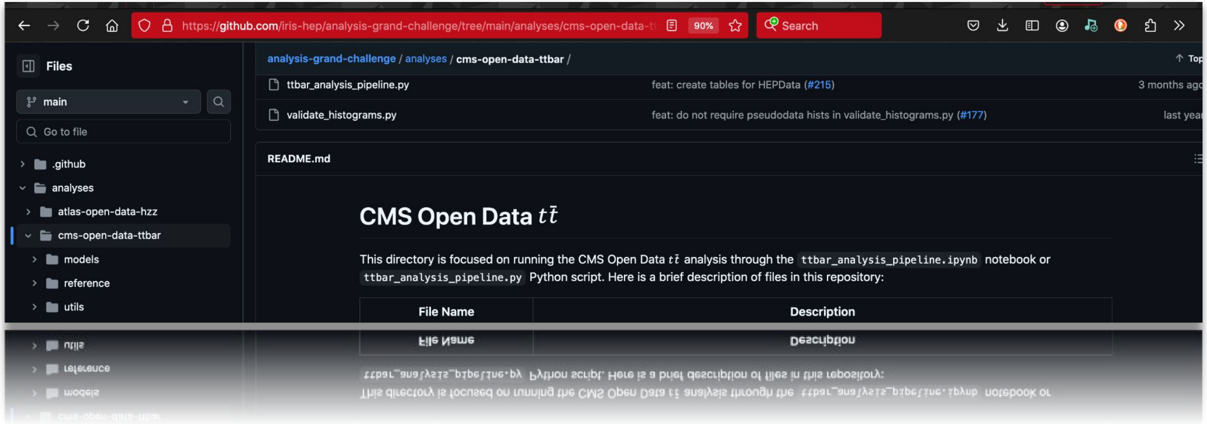
EO²C Analysis Facility

See also APPENDIX



Test cases: **CMS** Analysis Grand Challenge

https://github.com/iris-hep/analysis-grand-challenge/tree/main/analyses/cms-open-data-ttbar



	1	2	3	4
AGC Running modes	Single node **mt** multi-threaded	Single node **dask-local** multi-process	Multi nodes **dask-ssh** multi-process/node	Multi nodes **dask-remote** multi-process/node

CMS **AGC** Dataset & Formats

- Six datasets were used as input for the benchmarks, based on CMS OpenData *ttbar*, including **three generations** of **AGC RNTuple** files

Name	Format	Comp	Size	#Files
	TTree	ZLIB	1.94 TB	787
	TTree	ZSTD	1.59 TB	787
AGC^{1}	**RNTuple**	ZSTD	1.04 TB	787
AGC^{2}	**RNTuple** 2xcondensed Cluster Size 200M	ZSTD	1.04 TB	396
AGC^{3}	**RNTuple** Cluster Size 100M **Adaptive Pagesize**	ZSTD	965 GB	787
AGC$^{100\|200}$	**RNTuple** 100x inflated **AGC$^{1\|2}$**	ZSTD	104 TB	39600

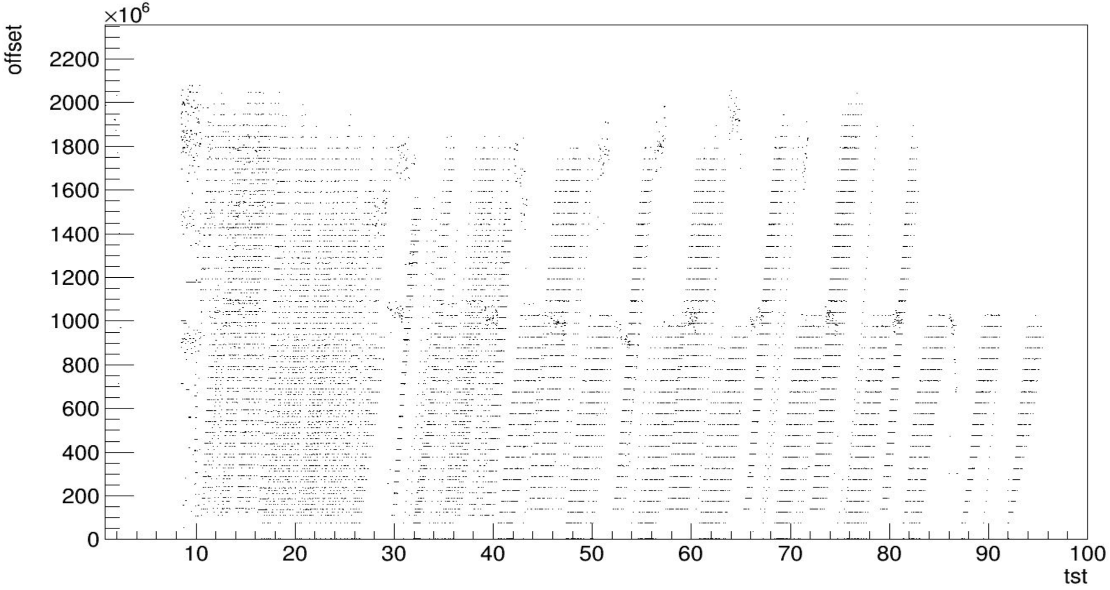
➔ The runtime of *ttbar* AGC is relatively short, making it unsuitable for running with distributed Dask and large-scale parallelism, *runtime* < *init.time*
(> 100 workers …)

➔ We created a 100x inflated Dataset adding each data file 100x into the **104 TB** **AGC$^{100|200}$** dataset

CMS **AGC** Read Pattern

- The CMS OpenData ttbar analysis poses a **challenging use-case** for a *spinning-disk-based* infrastructure because …

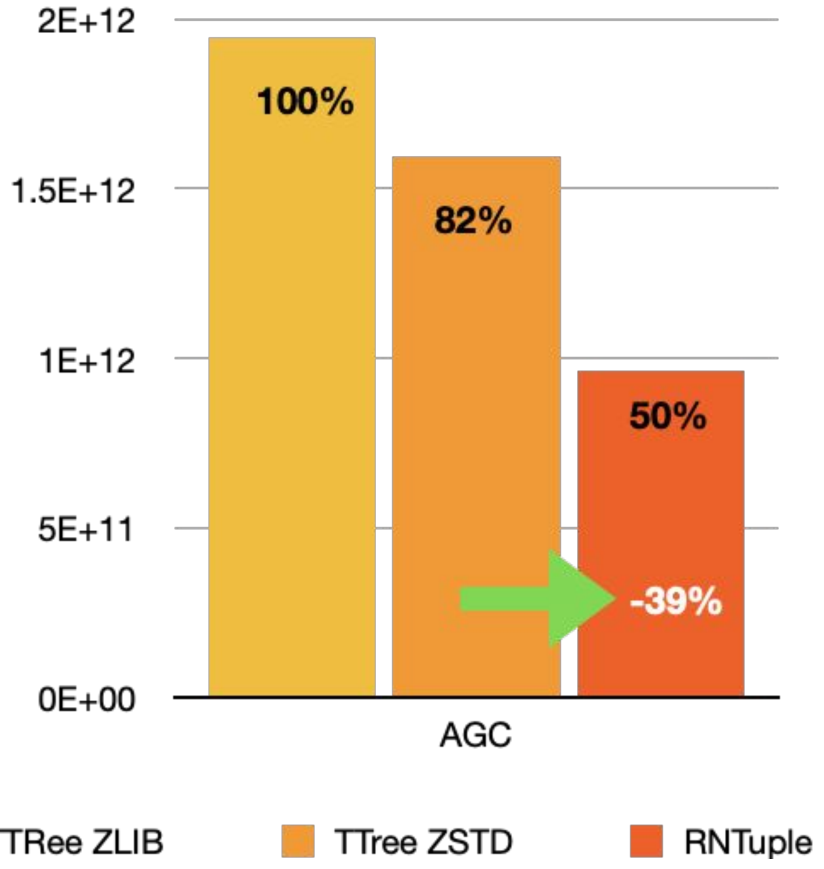
➔ the **analysis reads only 6.4% of the full data set** as input
- sparse scattered forward reading
- not really HDD friendly



AGC[1] ttbar read pattern may 2024 - x:timestamp y:file - offset

AGC RNTuple Size

Size Reduction from TTree to RNTuple



AGC[3]

Advantage of using RNTuple: the identical contents is stored using less disk space

	TTRee ZLIB	TTree ZSTD	RNTuple
AGC	1'946'631'920'767	1'594'321'501'163	964082593461000

➔ For a **fair comparison**, we rewrote the data using the ZSTD TTree format and compared the resulting size to RNTuple, achieving a 39% reduction in volume for **AGC[3]**

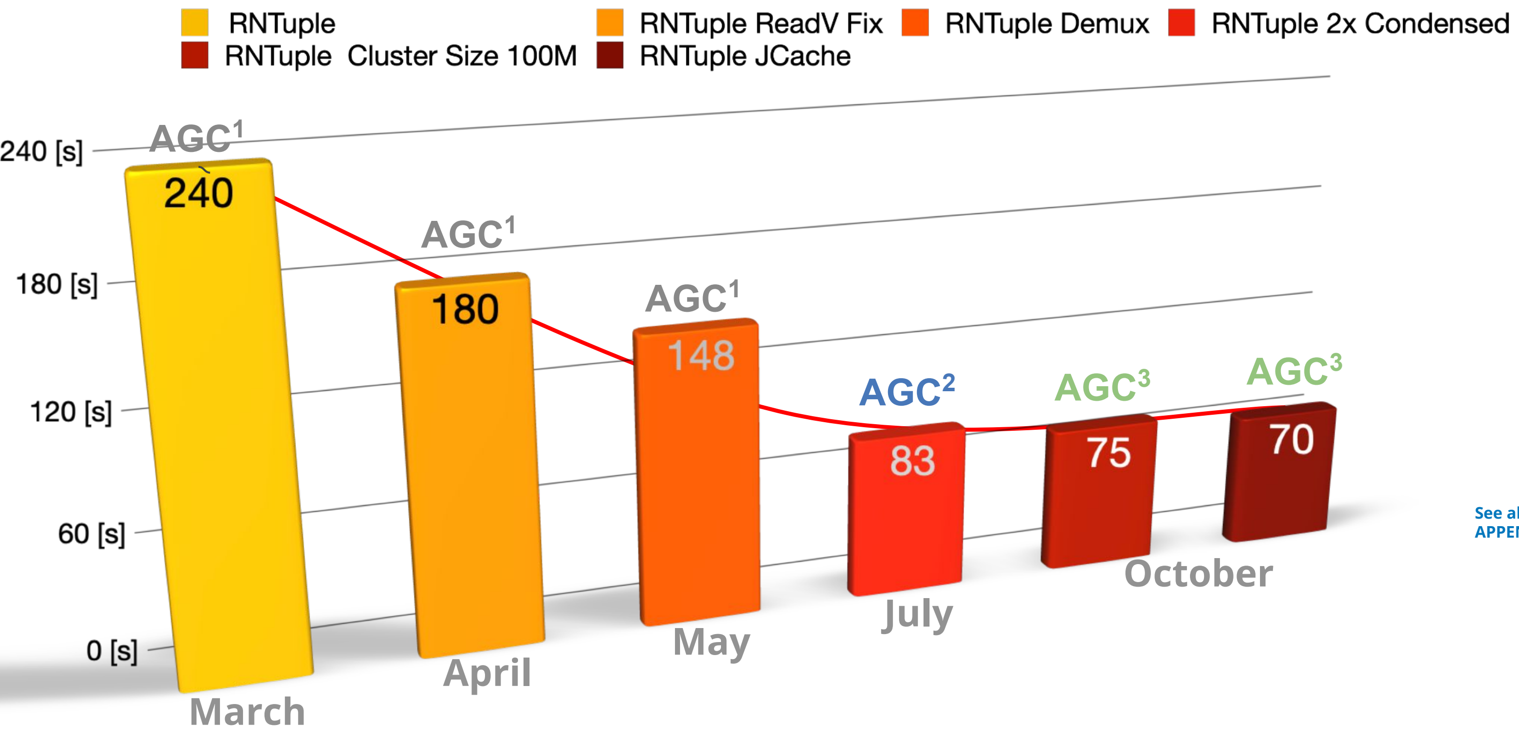
➔ How do Realtime & CPU when running the *mt* **AGC[1]** ttbar analysis**?**

Realtime Meas. March 2024:
TTree ZSTD : 250s
RNTuple AGC[1] : 240s
"Something is wrong …"

AGC mt

multi-threaded

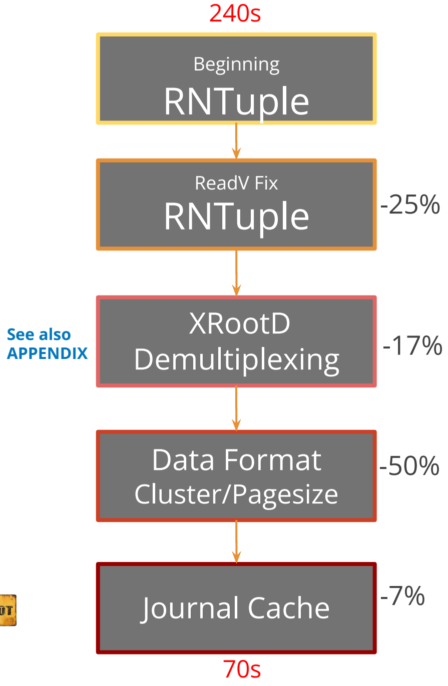
The AGC *mt* **RNTuple Journey** 2024 in short ~~240s~~→ 70s



Single Node AGC Benchmarking *32 cores* - ***multi-threaded*** - against **EOSPILOT**

Result:>**3X** faster

Holistic Optimization for Single Node with remote EOS

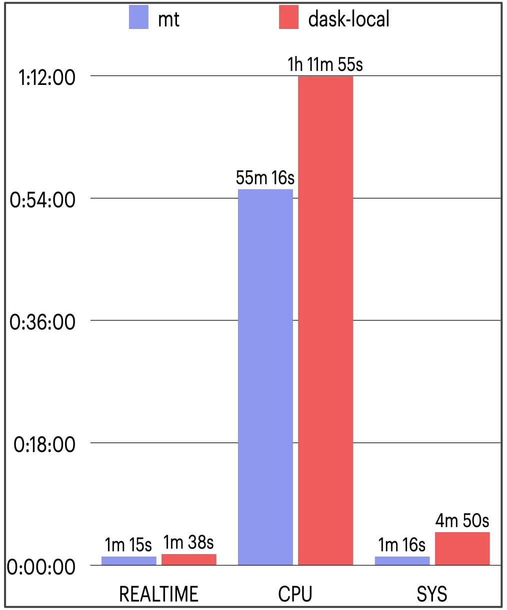


AGC dask-local

multi-process

The AGC **RNTuple Journey** 2024 - *mt* vs *dask*

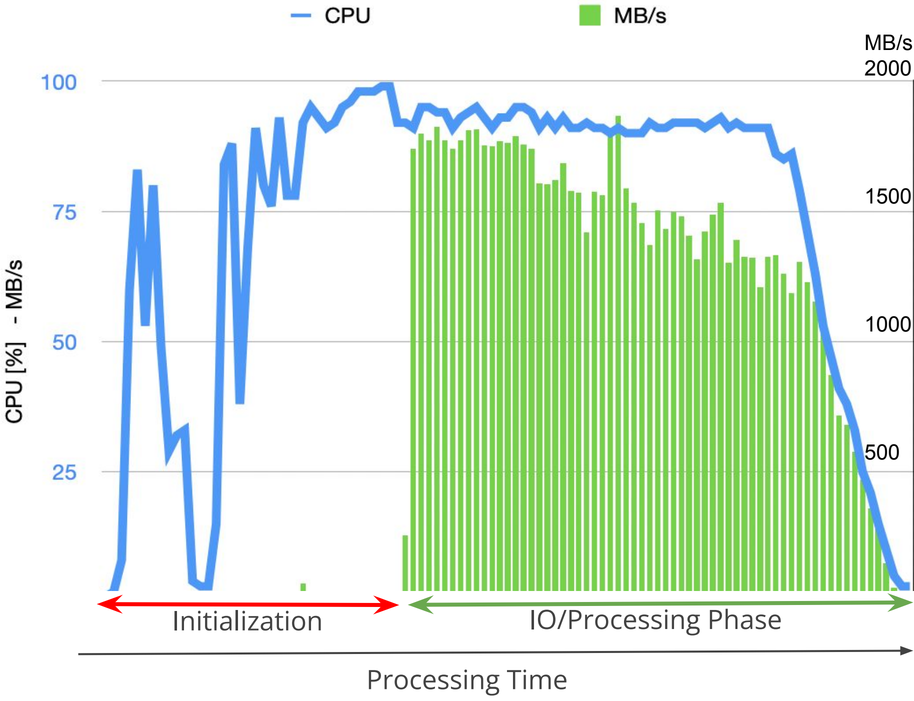
Figure: **Runtime** dask-local **AGC³**



- ***Green computing:*** *dask-local* uses ~30% more CPU than *mt* for the identical **AGC³** job
  - startup phase ⟶

- ***But****:* dask allows **scale out over multiple nodes**
  - startup phase low impact for longer running jobs

Figure: **CPU & IO** vs time for dask-local AGC

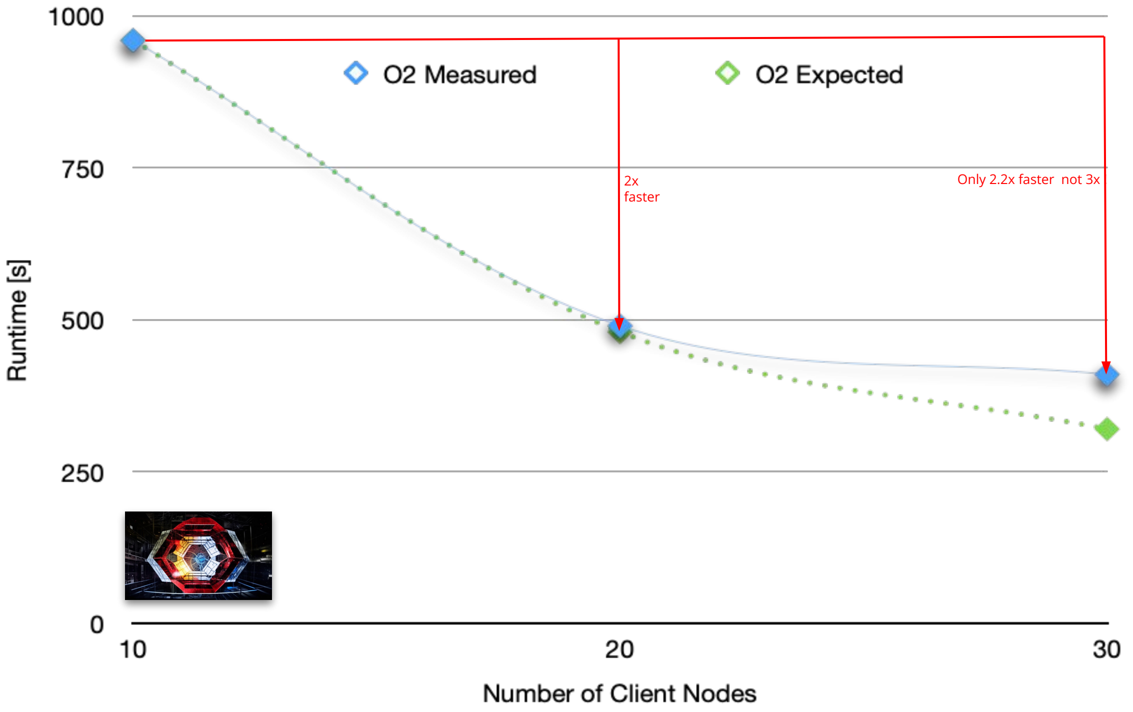


AGC dask-scale-out

multi-node/process

AGC **RNTuple** - IO scalability

- When scaling **AGC**^{100} to few nodes with *dask-ssh* it was able to saturate the **EOSPILOT** and even the **EOSALICEO^2** Instances … **not limited by bandwidth** …



- Performance scales up to 20 nodes, then scalability breaks with O^2 backend
- Each client node runs 64 dask worker
  - 30 nodes → 1920 worker

- Introducing modified RNTuple format for **AGC^{200}** with **EOSPILOT**

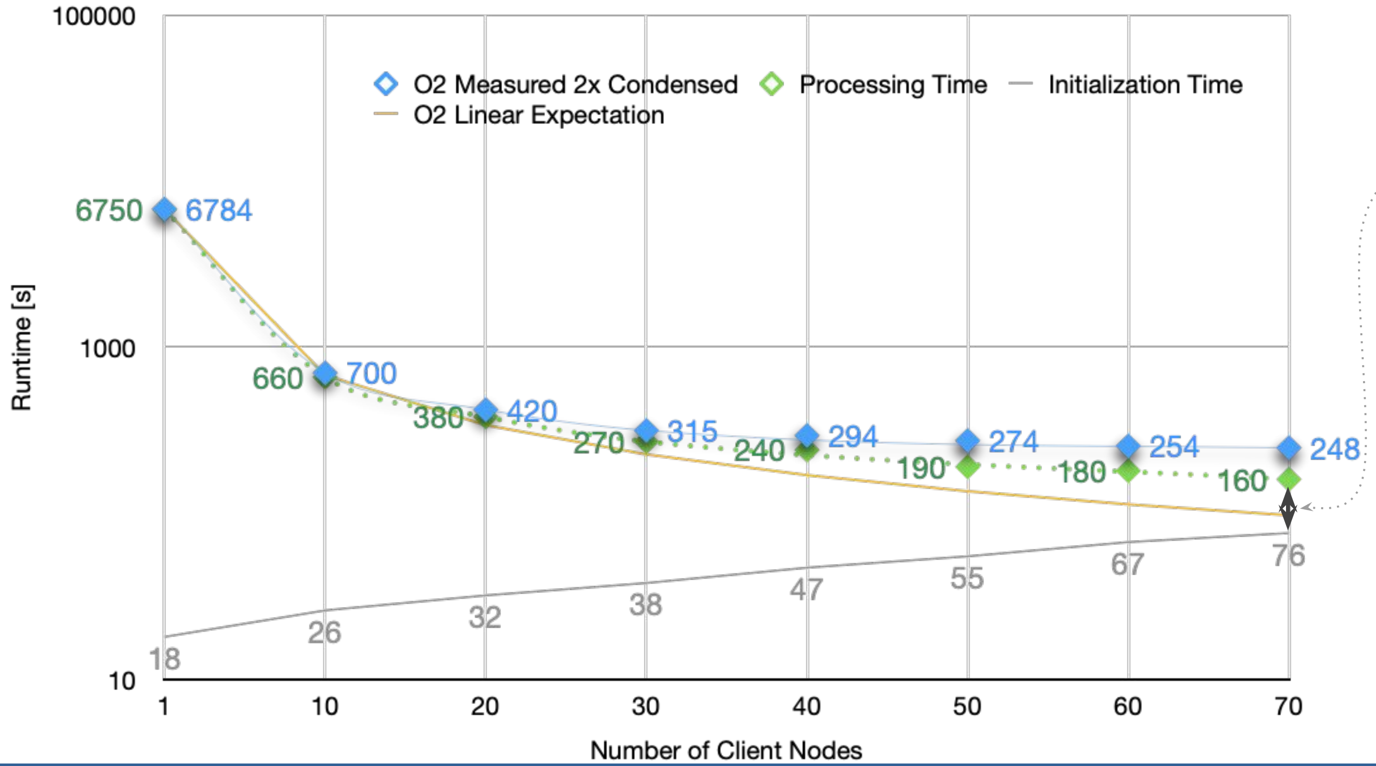
The **AGC^{200}** (condensed) data format resulted in significantly **larger and less** read requests:

~27kb →*~540kb*
~2M IOPS →*~100k IOPS*
787 files →*396 files*
reads are mostly contained in readV requests

It helped to boost the instance output using the same storage hardware by **factor >3** .

AGC **RNTuple -** Format Improvements - fewer **IOPS**

- Introducing modified RNTuple format for **AGC^{200}** with **EOSALICEO^{2}**



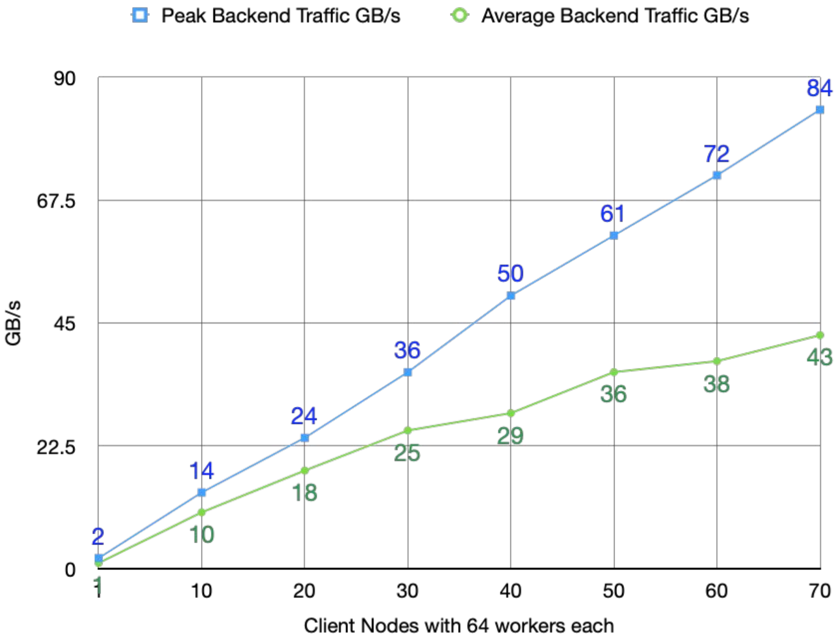
With a 100x inflated **AGC^{200}** dataset we observe that as the number of client nodes increases, the initialization time gets close the processing time, resulting in a breakdown of scalability.

Single Analysis
extremely sparse
reaches avg. INGRES
222 GBit/s
during processing
345 GBit/s

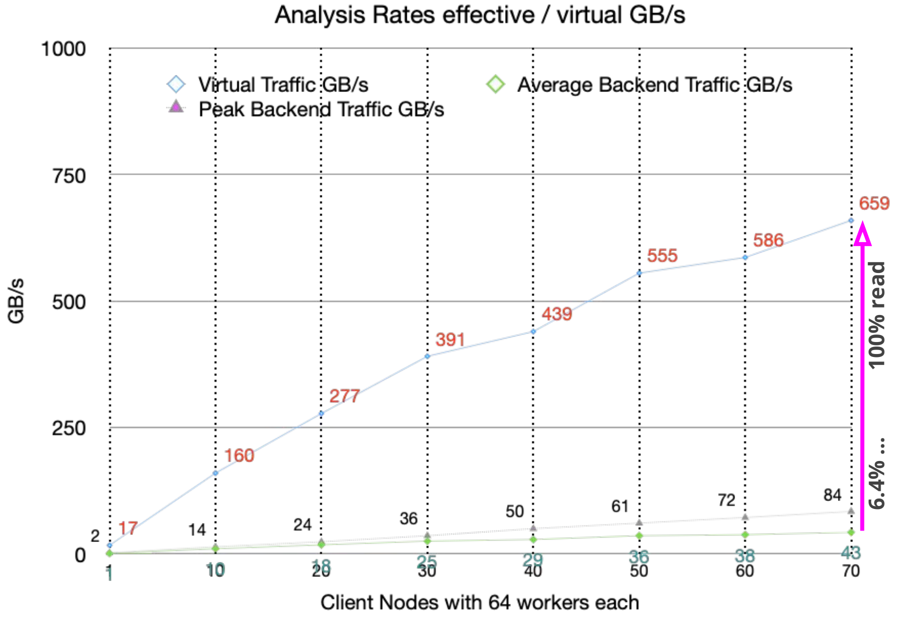
Side Remark:
Instance can do 5 TBit/s when streaming

AGC **RNTuple** Backend Traffic

- Average and peak output rates for 2x condensed RNTuple format for **AGC**[200] with **EOSALICEO**[2]



- **Virtual IO rates:** scaling traffic from 6.4% of the accessed data to the entire data - sparseness defines instance output [0.345 - 5 TBit /s]



"Can we use a high-performance shared filesystem as a distributed cache and still gain performance?"

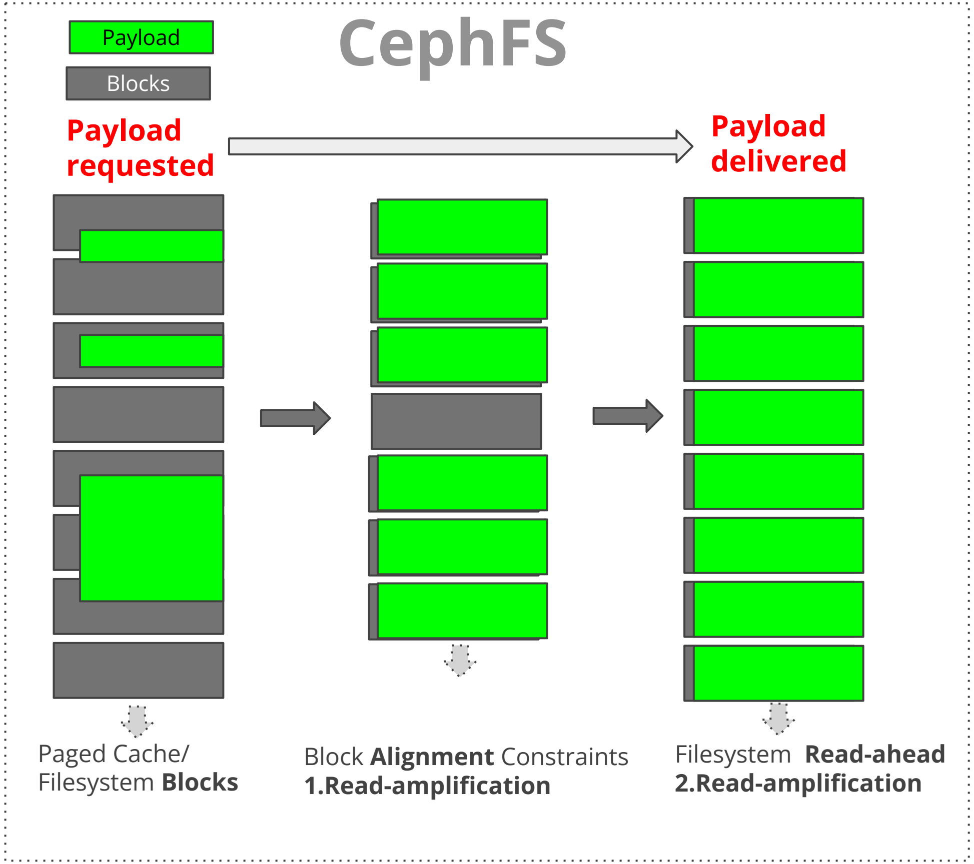
AGC Optimized Caching

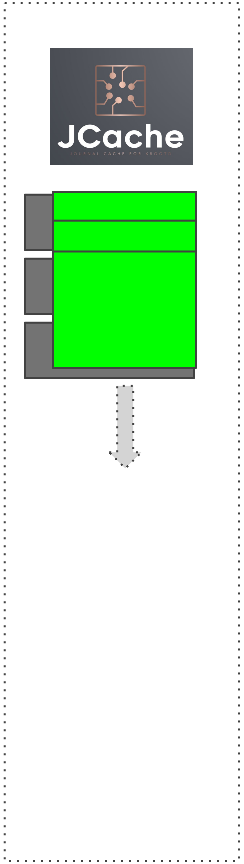
JCache

JOURNAL CACHE FOR XROOTD

R&D Project

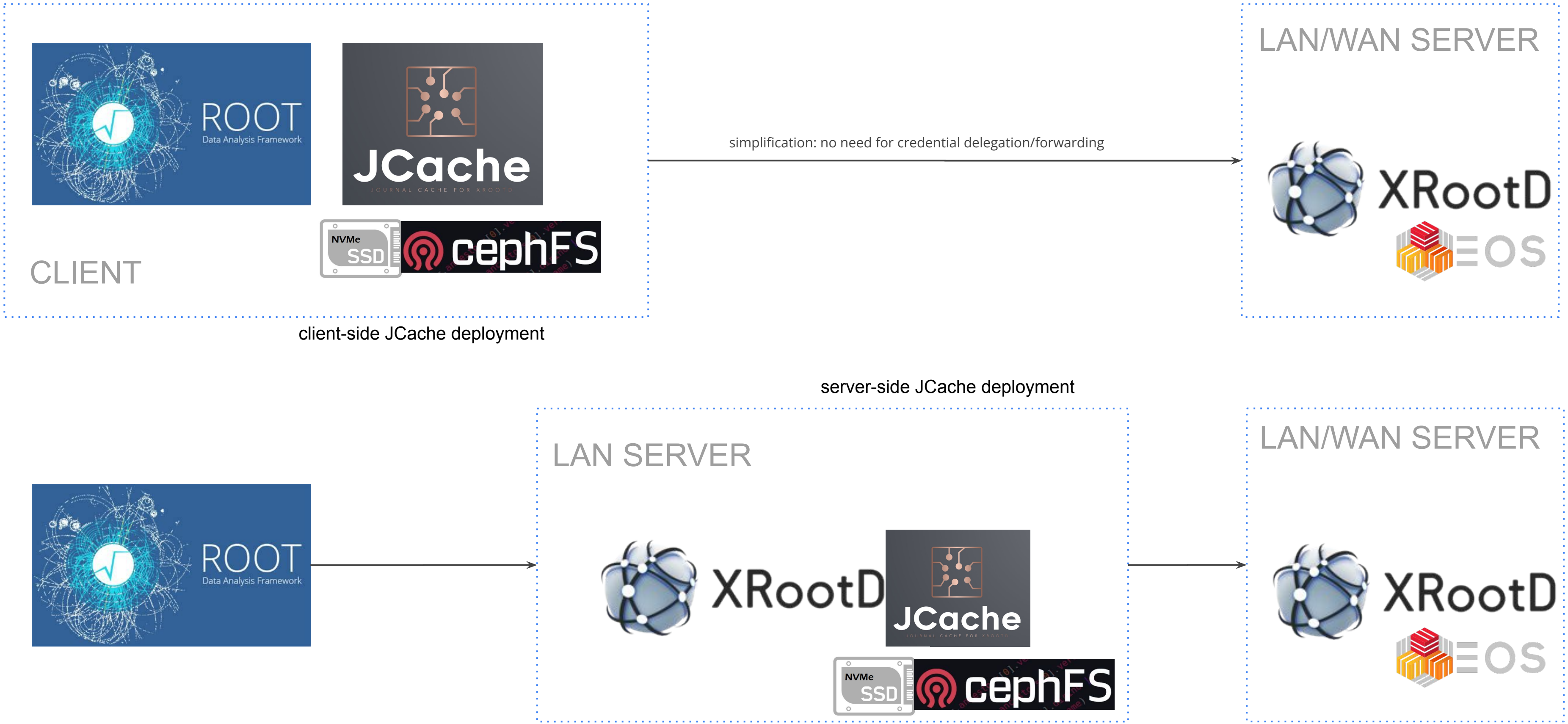
JCache - why journaling?





- By default **CephFS** creates **4x** read amplification
  - A **22.5 GB/s** Filesystem delivers **5.6 GB/s** of data you need
  - After disabling read-ahead still **+30%** amplification
- **JCache** avoids read amplification and space overhead in the cache
  - Works well with default read-head
  - **22.5 GB/s** used for the requested IO by applications

JCache - deployment model



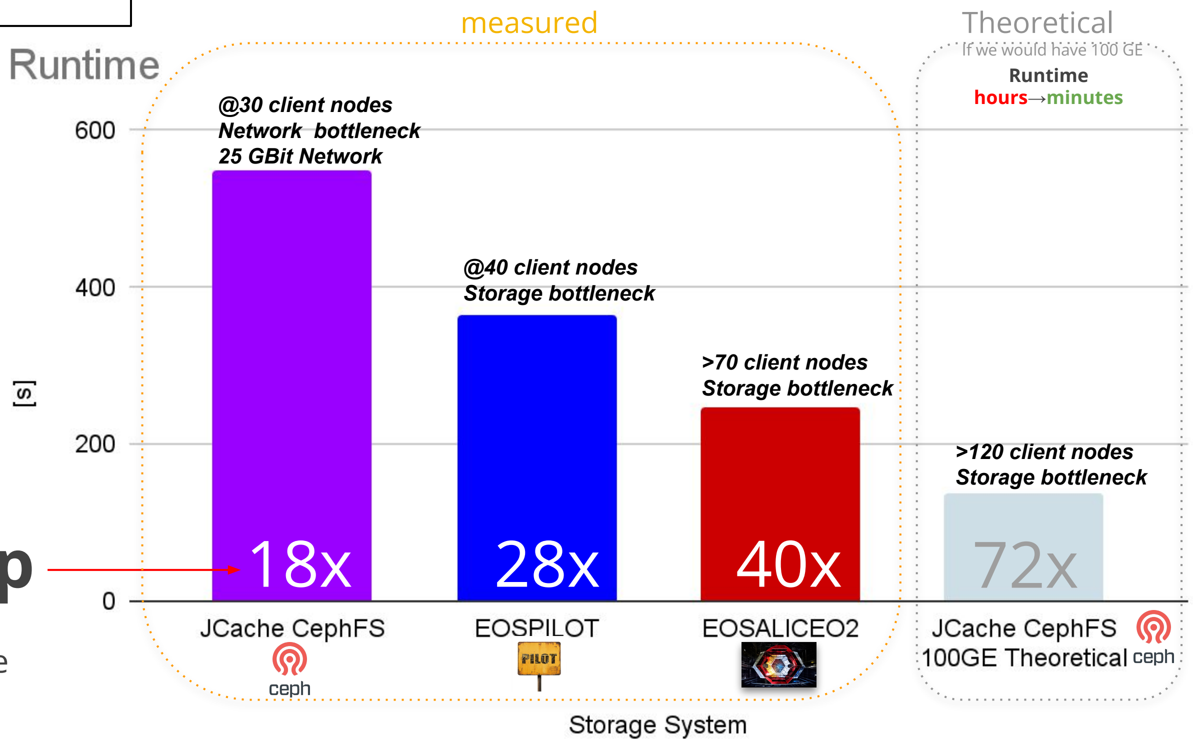
JCache - embedded IO benchmarking

```
# ----------------------------------------------------------------------- #
# JCache : 2024 CERN.EOS - Andreas-Joachim Peters                         #
# ----------------------------------------------------------------------- #
# JCache : cache combined hit rate  : 100.00 %
# JCache : cache read     hit rate  : 100.00 %
# JCache : cache readv    hit rate  : 100.00 %
# ----------------------------------------------------------------------- #
# JCache : total bytes    read      : 8976091232
# JCache : total bytes    readv     : 53958826610
# ----------------------------------------------------------------------- #
# JCache : total iops     read      : 34056
# JCache : total iops     readv     : 10707
# JCache : total iops     readvread : 93611
# ----------------------------------------------------------------------- #
# JCache : avg.  bytes    read      : 263568.00
# JCache : avg.  bytes    readv     : 5039584.00
# ----------------------------------------------------------------------- #
# JCache : open files     read      : 1050
# JCache : open unique f. read      : 796
# JCache : time to open files (s)   : 0.000
# ----------------------------------------------------------------------- #
# JCache : total unique files bytes : 976614396598
# JCache : total unique files size  : 976.61 GB
# JCache : percentage dataset read  : 6.44 %
# ----------------------------------------------------------------------- #
# JCache : app user time            : 3329.07 s
# JCache : app real time            : 72.48 s
# JCache : app sys  time            : 41.17 s
# JCache : app acceleration         : 45.93x
# JCache : app readrate             : 868.35 MB/s [ peak (1s) 1.94 GB/s ]
# ----------------------------------------------------------------------- #
```

- JCache provides easy insight into application IO including **bandwidth profiling**
- JCache is maintaining **100% async IO** and allows disconnected operation

```
# IO Timeprofile
# --------------
#  1670.00 MB/s |
#  1484.44      |        **  *
#  1298.89      |          ** **
#  1113.33      |       *       **      **   ***   **
#   927.78      |                  *  *  *       *  *
#   742.22      |      *          ** **    **   *
#   556.67      |                                *    * *
#   371.11      |
#   185.56      |                                      *
#     0.00      | ******
#                ----------------------------------------
#                0   10  20  30  40  50  60  70  80  90  [ 100 % = 72.48s ]
# -----------------------------------------------------------------------
```

What did we reach?



Speedup
compared to
single client node

- **Client & JCache Dimensioning** allows to **scale down run-time** "as you want" or "as you can afford"

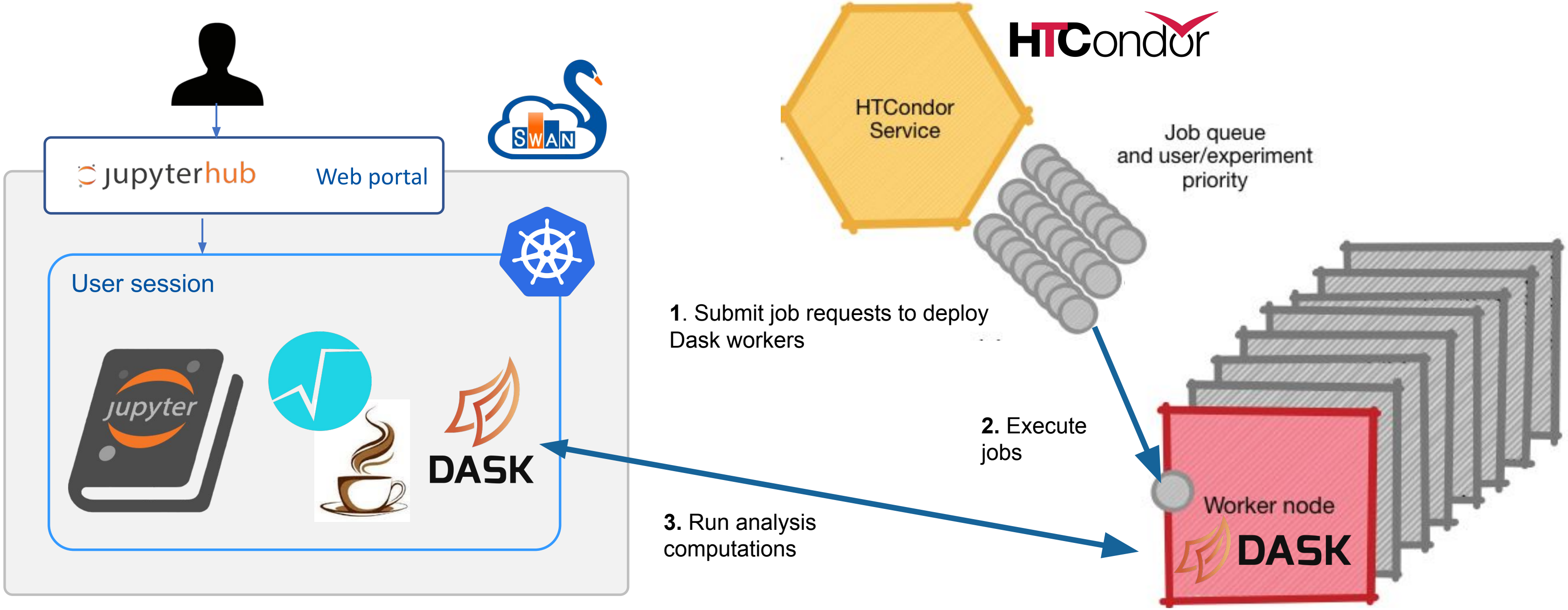
  "A solid strategy is to establish a data format that can be efficiently read from HDD back-ends."

AGC with SWAN&HTCondor*

*A Pilot Analysis Facility at CERN

dask-remote

CERN Analysis Facility Pilot - SWAN + HTCondor

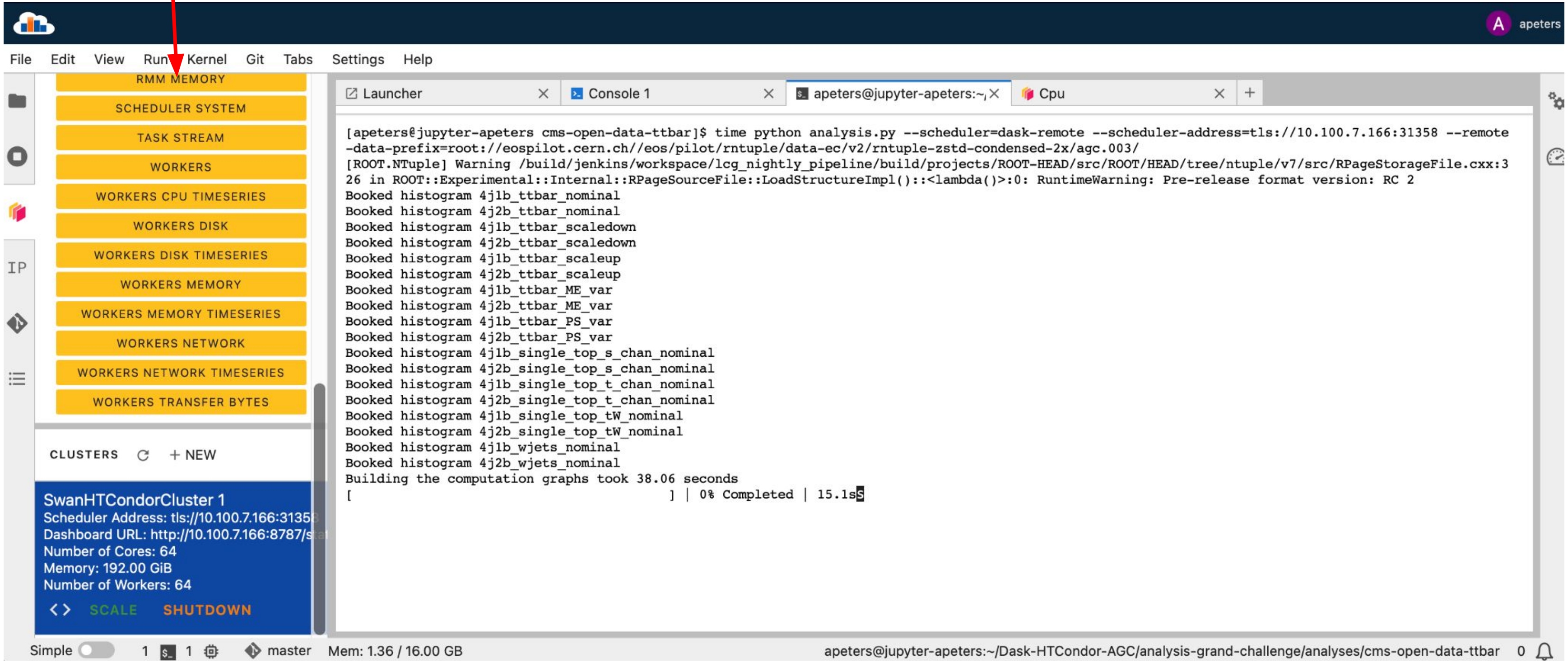


Courtesy: Slide from Enric Tejedor / CERN SWAN team

CERN Analysis Facility Pilot - SWAN + HTCondor

Extensive Monitoring Abilities

New Jupyter Lab Interface with HTCondor Plug-in



CERN Analysis Facility Pilot - Results

- **Positive**
  - **Works** with similar runtime - assuming a similar performance environment for Dask workers operating in batch
  - **Simple** access in a web browser
  - **Integration** of EOS/CERNBox Sharing
  - **Access to hundreds of thousands of cores** in batch farm
- **Room for Improvements**
  - **Long time to initialize setup** for an interactive system
    - can take few minutes
  - HTCondor cluster **interface can get *disconnected***
  - **Dask initialization time** correlated to number of workers
    - User needs to understand how many workers are useful for a given task
  - The **setup is not yet fully automated** and requires few manual steps

AGC cms ttbar benchmark

More about the SWAN+HTCondor analysis pilot this afternoon see https://indico.cern.ch/event/1338689/contributions/6010680/

Summaries

Chapter 2 - Summary

Holistic Approach **RNTuple** + **EOS**

- **Detailed benchmarking** has allowed for **substantial improvements** in RNTuple for remote access from EOS (AGC example)
  - Excavated several issues which don't appear in local environments
  - Achieved **>3x** less run-time for TTree → RNTuple
  - Achieved **-39%** size reduction for TTree → RNTuple
  - Achieved **-40%** size reduction for moving from Replication → EC(10+2) in EOS
  - Achieved running extreme case of sparse CMS AGC analysis at **345 GBit/s** during processing (HDD storage)
- **Optimized caching** plug-in R&D **JCache**
  - Suitable to use high-performance shared file systems as shared cache with low overhead
  - Provides out of the box IO summaries and profiles
  - Allows to optimize AGC benchmark to lowest run-time performance eliminating read overhead
  - Suitable to run on end-user devices
- These results can provide **valuable input for the architecture of future analysis facilities**
  - NVME and HDDs - Shared File System combined with Remote Accessible Storage
  - Baremetal approach and/or SWAN+HTCondor
  - Usability and savings with EOS erasure coded storage for analysis

Final Summary & Future Outlook

A special thanks to all the teams from the IT and EP group who have helped advancing this project.

Author List

Andrea Sciabà (CERN)
Andreas Joachim Peters (CERN)
Florine de Geus (CERN/University of Twente (NL))
Guilherme Amadio (CERN)
Jakob Blomer (CERN)
Jonas Hahnfeld (CERN & Goethe University Frankfurt)
Markus Schulz (CERN)
Philippe Canal (Fermi National Accelerator Lab. (US))
Vincenzo Eduardo Padulano (CERN)
Alaettin Serhan Mete (Argonne National Laboratory (US))
Danilo Piparo (CERN)
Matti Kortelainen (Fermi National Accelerator Lab. (US))
Giacomo Parolini (CERN)

- **RNTuple** is transitioning to production
- Solid basis for **future IO** R&D
- Local and remote access **optimizations with significant improvements**
  - used a worst-case test case validating NVME and HDD storage backends
- For the final part of this activity we **count on participation from experiments** to
  - optimize specific data formats
  - validate them in a large scale data challenge

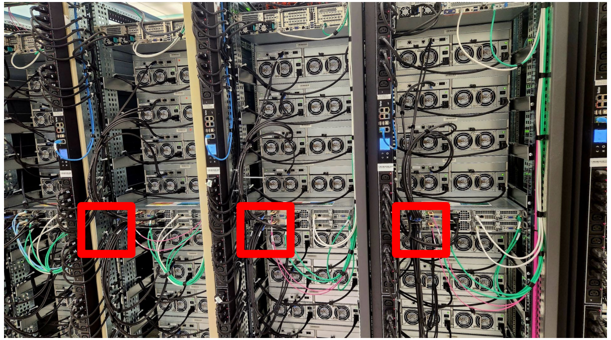
Thank You For Your Attention!

Appendix

Storage & Compute Hardware

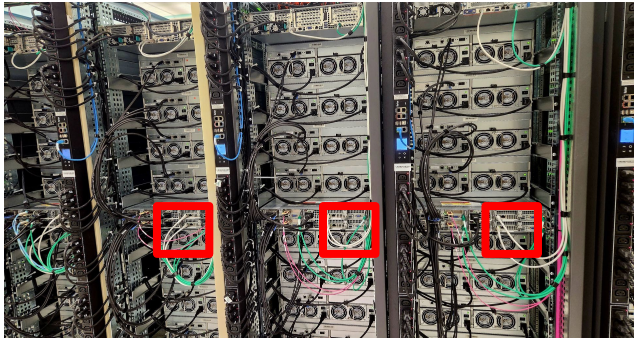
Storage EOSPILOT/ALICEO2

- **2x AMD EPYC 7302 and 7313** 16-Core Processor
- DDR4 3200 MT/s 16x16 GB - **256 GB**
- Intel® Ethernet Network Adapter E810 - **1x100GE**
- Filesystems
  - / EXT4 2 TB NVME
  - **/data01..96** XFS 18TB HDD



Compute EO^{2}C

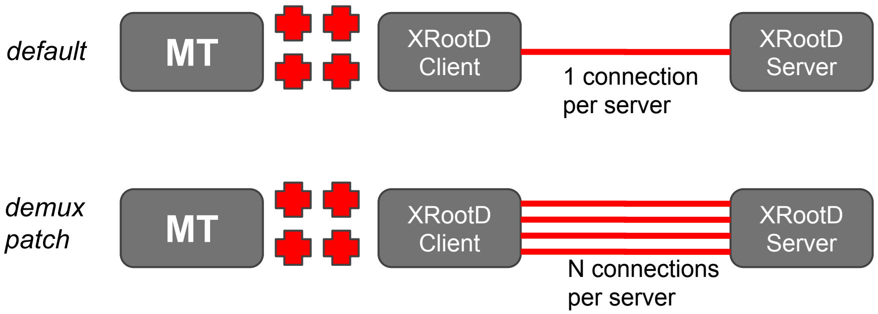
- **2x AMD EPYC 7302 and 7313** 16-Core Processor
- DDR4 3200 MT/s 16x16 GB - **256 GB**
- Intel® Ethernet Network Adapter E810 - **1x100GE**
- Filesystems
  - **/** EXT4 2 TB NVME
  - **/cvmfs** CvmFS filesystem
  - **/shared** CephFS home directory
  - **/jcache** CephFS cache directory



Single Node multi-threaded AGC Benchmarking *32 cores*

Observation: when you repeat a measurement twice reading initially uncached data from remote you see a significant reduction in run-time with the second run. This is due to a client side bottleneck!

Connection Demultiplexing eliminates this bottleneck.



AGC[2]

Mode	1st run	2nd run
Default	142s	77s
Demux[64]	83s	75s

- Reported last year at CHEP in the context of **XCache benchmarking**
- If your storage is made of **more servers than threads used** in your application, multiplexing does not play a role (e.g. large CERN instances like EOSATLAS/CMS/PUBLIC)
- For **EOSPILOT** it still plays a role.
- XRootD team identified problem in parallel-socket implementation - when this is fixed enabling parallel sockets should eliminate this problem without the need for *manual multiplexing*

Final Sprint Improvements - from **XCache** to **JCache**

Can we use an NVME based shared filesystem as a distributed cache and gain performance?

yes! *R&D project*

- **XCache**
  - well established in HEP as a block based XRootD based cache server
    - If data is only partially accessed block chunking introduces **moderate to significant space overhead** of the payload vs cached data contents e.g. 128k blocksize +123%
    - there is also a visible **real time overhead** when reading cold data through XCache in a LAN environment with not saturated infrastructure (***2x - 5x*)**.
    - Implemented as a server-side plug-in in XRootd
- **JCache**
  - JCache is using **journaling** to cache file contents not blocks (code derived from **eosxd**)
    - Stores only payload - **0% size overhead** - **15% real-time overhead** for cold AGC case in LAN
    - Implemented as a client plug-in which can be deployed client and server-side

JCache - caching into shared filesystems - CephFS

- Using **file locking** for cache journals allows to use a **shared file system as a distributed cache** on many clients
  - we used CephFS with 8 NVME nodes (see before) and double replication

- One drawback of using a shared file system is the impact of **automatic read-ahead and object block sizes** defined
  - With default mount options **CephFS inflates** the required **traffic** on the wire **>4x** due to the automatic read-ahead algorithm
  - With **ra=64k or ra=0k** this creates still **1.3x more traffic** than required
  - We used a default block size of 4M, tried also 1M - no change in behaviour

- When running **AGC**^{200} on 30 nodes using dask-ssh the **run-time** is defined by the **maximal bandwidth** of the CephFS backend and the read **overhead**
  - The CephFS back-end saturates at around **22.5 GB/s**
    - Run-time is 550s - 50% longer than on EOSPILOT (due to bandwidth limit + read overhead)
    - Origin is that our CephFS cluster has only 25GE on each storage node!