RNTuple tests

Datasets (old)

- AGC dataset in TTree format compressed with ZSTD:
  - `root://eospublic.cern.ch//eos/root-eos/AGC/ttree-zstd/` (original)
  - `root://eospilot.cern.ch//eos/pilot/rntuple/[data-ec | data-rep]/agc.NNN/agc/datasets/ttree-zstd/` (10 replicas)
- AGC dataset in RNTuple format:
  - `root://eospilot.cern.ch//eos/pilot/rntuple/[data-ec | data-rep]/agc.NNN/agc/datasets/rntuple/` (10 replicas)
- EOS file replication: `data-rep`
- EOS erasure coding: `data-ec`

Datasets (new)

- Master copy of the datasets usable for tests:
  - `root://eospublic.cern.ch//eos/root-eos/AGC/ttree-zstd/nanoAOD`
  - `root://eospublic.cern.ch//eos/root-eos/AGC/rntuple/nanoAOD`
- Copies in EOSPILOT:
  - `/eos/pilot/rntuple/data-ec/v2/ttree-zstd/agc.<n>/nanoAOD`
  - `/eos/pilot/rntuple/data-ec/v2/rntuple/agc.<n>/nanoAOD`
  - `/eos/pilot/rntuple/data-ec/v2/rntuple-zstd-condensed-2x/agc.<n>/nanoAOD`
  - `/eos/pilot/rntuple/data-rep/v2/ttree-zstd/agc.<n>/nanoAOD`
  - `/eos/pilot/rntuple/data-rep/v2/rntuple/agc.<n>/nanoAOD`

Preliminary measurements

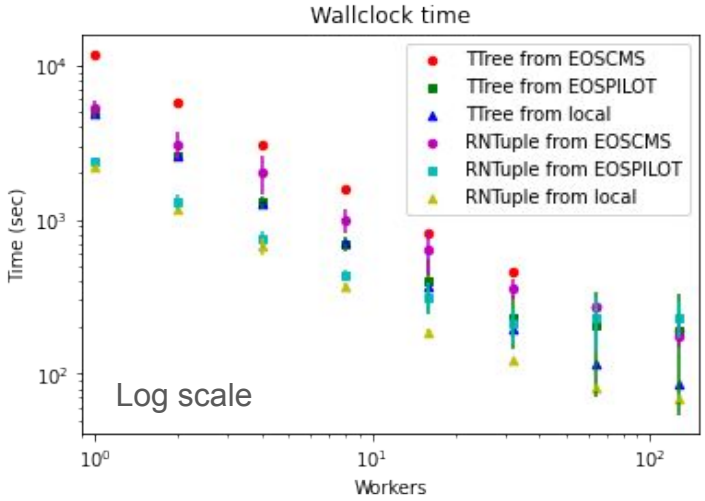
- Conversion time from TTree to RNTuple for a random NANOAOD file from the AGC dataset
  - 0.10 s/MB
- Dataset sizes
  - TTree: 1.77 TiB, zlib compressed
  - TTree: 1.45 TiB, zstd compressed
  - RNTuple: 1.04 TiB, zstd compressed

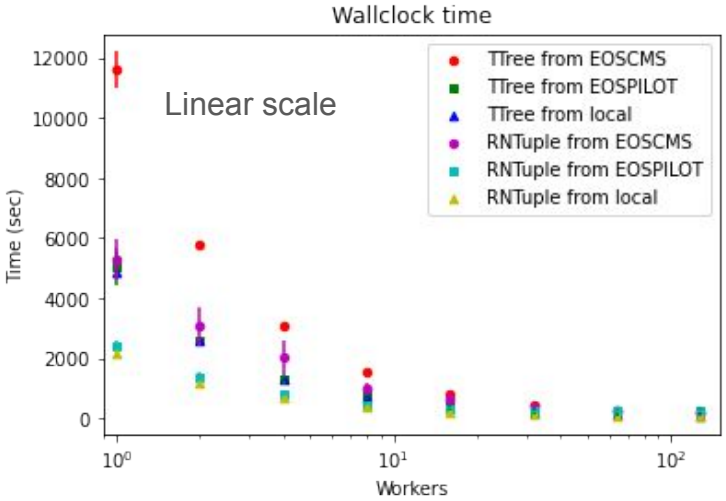
First TTree - RNTuple comparison

- AGC workload for RDF (link)
- Original nanoAOD dataset in TTree format
  - `root://eoscms.cern.ch//eos/cms/opstest/asciaba/agc/datasets/nanoAOD`
  - `root://eospilot.cern.ch//eos/pilot/rntuple/agc/datasets/nanoAOD`
  - `/data/datasets/agc/datasets/nanoAOD` on *iopef01.cern.ch*
- Dataset converted to RNTUple using provided tool (link)
  - `root://eoscms.cern.ch//eos/cms/opstest/asciaba/agc/datasets/nanoAODRNTuple`
  - `root://eospilot/cern.ch//eos/pilot/rntuple/agc/datasets/nanoAODRNTuple`
- For the conversion I used a ROOT nightly build from 15 January
- To run the workload I used a ROOT nightly build from 29 January
  - Needed to resolve crashes with the previous version
  - Cannot use XRDRecorder with the RNTuple dataset due to an unknown XrootD bug

Wallclock time and scalability

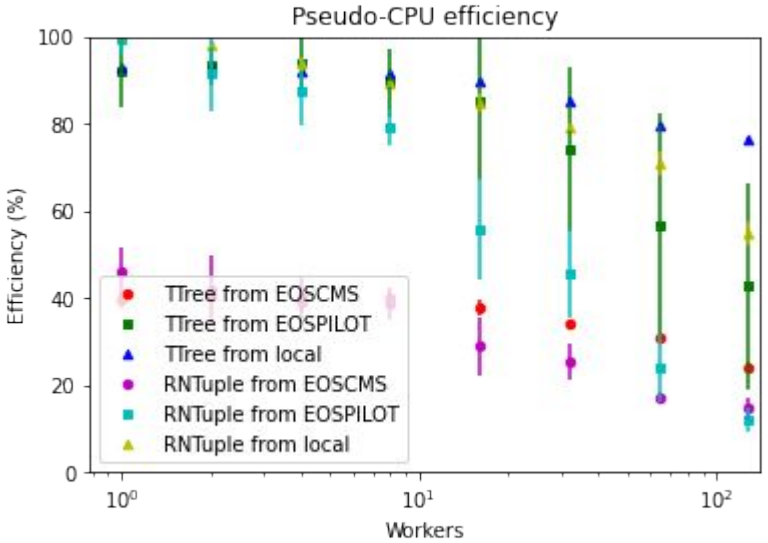
- RNTuple jobs 2x faster than TTree for low number of threads!
- EOSPILOT 2x faster than EOSCMS faster for low number of threads!
  - Almost as fast as local access
- Scalability poor for many threads for EOS, still very good for local access





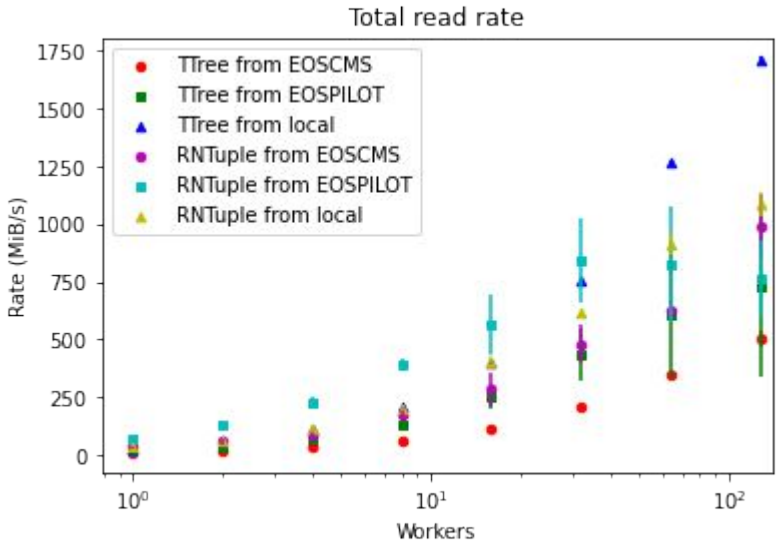
CPU efficiency

- CPU time / (wallclock time * ImplicitMT threads)
  - Actual number of threads is much larger
- CPU efficiency good for local access, bad for EOS access and slowly decreasing with number of threads
- Huge variations between runs for EOSPILOT
  - Sensitive to other activity or server-side caching?



Read rates

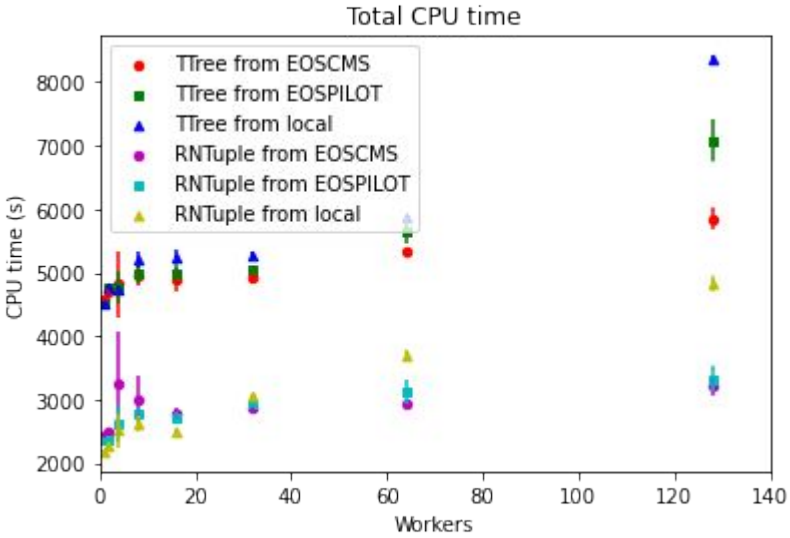
- Local read rates higher for TTree
  - Much more data is read!
- XRootD read rates higher for RNTuple
  - Much more data is read!
- Total data read changes depending on source and format
  - RNTuple reads less from local, the opposite for TTree. Why?
- Checked that total data read does not depend on the number of threads



	Total data read (GiB)	
	EOS	Local
TTree	91	142
RNTuple	165	73

CPU time

- CPU time is significantly less when reading from RNtuples
  - Decompression faster?
- CPU time does not depend (much) from the source (EOS or local)



Conversion speed

- The performance of the TTree-RNTuple conversion was measured using one process per file, with up to 32 processes per node and up to 77 nodes
- Ten copies of the AGC dataset (with ZSTD compression) are used, located on EOSPILOT data-rep area
- Output is written to EOSPILOT data-rep area, in a different directory for each node
- Prmon is used to monitor each conversion process and to measure
  - CPU efficiency
  - Average read rate (file size divided by wallclock time) per process
  - Average write rate (file size divided by wallclock time) per process
  - Network read and write rate per node
- Results for:
  - 8 processes/node, 1 node (TODO)
  - 32 processes/node, 1 node
  - 32 processes/node, 77 nodes

Results

Number of nodes	Process read rate (MB/s)	Process write rate (MB/s)	Process CPU efficiency (%)	Node network read rate (MB/s)	Node network write rate (MB/s)
1	9.2 ± 1.2	6.2 ± 0.2	93 ± 3	287.7	189.9
77	8.2 ± 3.2	5.2 ± 2.2	81 ± 8	251.1 ± 1.1	165.1 ± 1.1

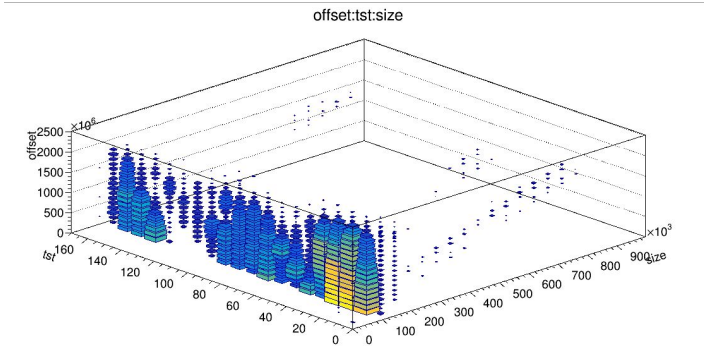
- Whole node rates are ~32x the process rates, as expected
- 15% performance degradation when going from one to 77 nodes
  - Where’s the bottleneck?
- Conversion processes fairly efficient
- Total rates to/from EOSPILOT should be about 19 GB/s and 13 GB/s respectively
  - To cross-check with monitoring

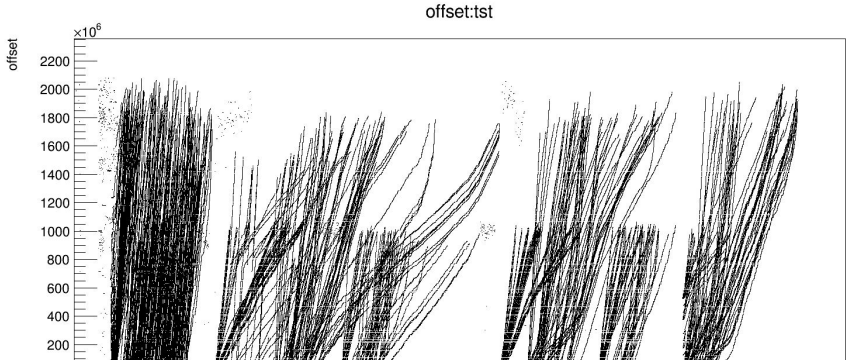
Timeline

- 3/4 (GA): need to fix recorder plugin. To test on all combination of (replicated data, EC data) and (TTree, RNTuple)
- 11/4 (GA): tested recorder plugin with devel XRootD and ROOT 6.30.06, no crashes (16 cores, 32 threads) (results)
  - Synchronicity(R): 97%, not good, but forgot to use python3.12 -X perf
  - TODO: test with XrootD 5.7.0

- 12/4 (AP): measurements of AGC code with EOSPILOT, results are easily affected by the files being cached or not!
  - Baseline
    - TTree-ZSTD, MT: 105 s (wtime), 5200 s (cputime), 1600 MB/s reading from local NVMe
      - Reading remote with data cached: 110 s (wtime)!
    - Reading remote with uncached data, EC files on 32c node:
      - TTree standard dataset: 1.95 TB, 160 s (wtime), 4040 s (cputime), 600 MB/s
      - TTree ZSTD dataset: 1.59 TB, 170 s (wtime), 4482 s (cputime), 1000 MB/s
      - RNTuple dataset: 1.14 TB, 320 s (wtime), 1900 s (cputime), 200-800 MB/s!
    - With replicated files (data served by 1 HDD instead of 10-12):
      - TTree: 540 s (wtime)
      - RNTuple: 300 s (wtime)
  - To summarize, RNTuple takes 72% of size wrt ZSTD TTree, takes 2x the wallclock time, reads 2x the volume and uses half the CPU time
  -

- 15/4 (AP): Compared the recorder plugin output for TTree standard and TTree ZSTD. In general, the numbers are similar but +10% IOPS with ZSTD!!
- 15/4 (AP): With TTree, AGC reads about 94 GB while it reads about 170 GB for RNTuple





- 16/4 (AP): AGC on RNTuple local takes 84 s and reads only 67 GB, but reads 170 GB when reading remote. Summary of observations:
  - RNTuple files are significantly smaller
  - wtime when reading locally is good if NVMe does 1.5 GB/s on 32c node (data cached or not is the same)
  - wtime for reading remote is much worse than TTree [NOT WHAT ANDREA SEES]
  - Data read from wire is 2.5 times than locally
  - RNTuple remote uses 1280 kB reads, too small for WAN access, and no vector reads
  - Reading RNTuple files from a single XRootD server sucks because of the small serialized reads on the single connection
- 16/4 (GA): found out that recorder plugin does not support some kinds of operations (e.g. writev, xattrs), so it's incompatible with the XrdEc plugin. The plugin is now fixed

- 16/4 (JB): the I/O scheduler is identical for remote and local files, but found a bug in the RNTuple XrootD client code!
- 16/4 (JN): two issues:
  - Reading too much data with XRootD: due to a ROOT internal caching layer accidentally turned on (fixed by this PR)
  - Low number of vector reads: caused by the RNTuple prefetcher not setting a good prefetch window. Triggered by the first issue because only scalar reads are affected by the 128 kB cache (fixed by this PR)
- 16/4 (AP): RNTuple ROOT should read only the data that is needed, not doing a blind read-ahead. This is already the case for TTree (which also does vector reads!) and it must be the same for RNTuple. Right now, a lot of bandwidth is getting wasted for no good reason.
  - When using HTTP as protocol, things are MUCH better!

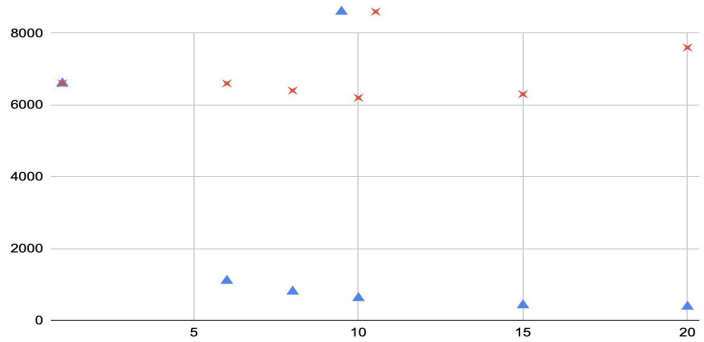
- 19/4 (AP): after the fixes, now performance with XRootD is the same as with HTTP! Wtime is 156 s uncached and 97 s cached with EOSPILOT
- 19/4 (DP): it is not possible to overcommit the threads (maximum is the number of hardware threads)
- 22/4 (JB): this is how parallelism works:
  - ROOT will use a given number of threads (max is no. of hw threads)
  - Each thread works on one file at a time, and typically each thread will work on a different file (unless the threads are more than the files, but even so each thread has its own file descriptor)
  - Every compute thread has a background thread for the I/O, doing the vector reads, filling a read-cache with page data in parallel to the processing
  - Every thread will open the next file when working on the current file, so each compute thread will typically have two open file descriptors. The file opening is not yet done in the background, though
  - The readv operations are asynchronous, BUT each thread only makes one readv at a time
- 22/4 (JH & AP): please note that the recorder plugin will record Synchronicity(R)=100%, as the parallelism is on the application side, not in XrootD, and it is about reading in parallel different files!

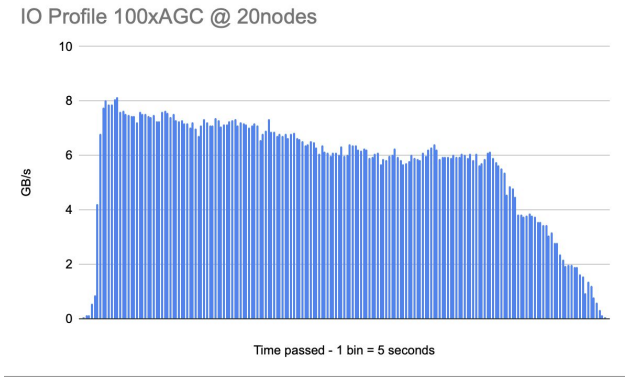
- 23/4 (AP): running on a cached RNTuple dataset takes 96 s and +60 s if uncached. With the TTree+ZSTD dataset, reading takes longer than RNTuple:
  - Cached: 147 s
  - Uncached: 204 s
  - Propose to add a way to tune the pre-fetching size
- 30/4 (AP): AGC using MT when scaling to many threads exhibits a highly fluctuating read rate and a large difference when the data are cached or not, while scaling out via Dask shows a very stable read rate with no difference for cached vs. uncached (and it's faster than MT, 100 s vs. 240 s)
- 30/4 (AP): files get open at the beginning and never closed until the end! JB confirms that this is not right, but the reason why it's done is because RDF could run multiple loops on a file. Not sure how this should be optimized
- 30/4 (VP): added an env variable (ROOT_RNTUPLE_CLUSTERBUNCHSIZE) to adjust the cluster bunch size

- 3/5 (AP): using Andrea’s code to fake many clients to have only one connection per core, uncached runs give 106-110 s consistently. This is because also EOSPILOT has only few servers and request serialisation has an impact. Still it remains the fact that Dask is faster than MT for the user even if it uses more CPU
- 3/5 (AP): compared reading from a RAID0 NVME node via 100GE link, using either a connection/thread or a single XRootD server process:
  - One connection/thread: 34 threads: 93 s, 64 threads: 86 s
  - One server process, 32 threads:
    - Block size 128k: cold 195 s, warm 97 s, final cache-size 197 GB
    - Block size 4k: cold 477 s, warm 93 s, final cache-size 88 GB

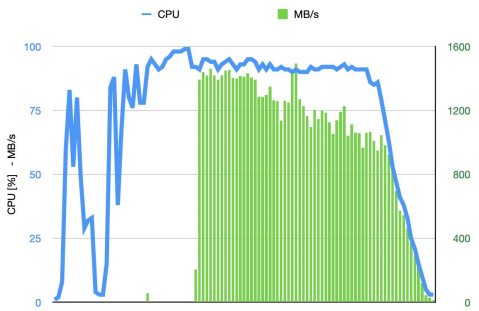
- 7/5 (JB): explains that RDF objects keep the files they need open until destruction, to avoid re-opening files unnecessarily, and Python might even defer the destruction. He puts a pull request to close the files at the end of the event loop for long chains (more files than threads)
- 16/5 (AP): creates a 100x dataset by replicating the files of the original dataset

- 16/5 (AP): made scalability tests using the 100x dataset. In the plot, the X axis is the number of workers (32 cores each) and the Y axis is runtime in blue and renormalized runtime for a single node (blue x nodes) (red)
- The second plot is the I/O profile with 20 nodes
- With the AGC access pattern, HDDs can make only 8-9 MB/s compared to a maximum of 260 MB/s

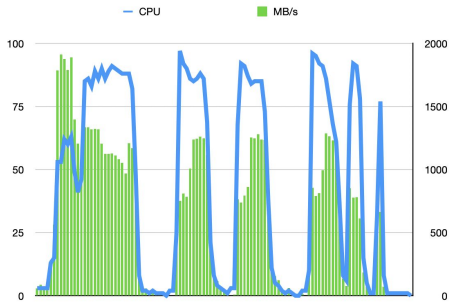




- 28/5: we had a meeting
- 30/5 (VP): produced new versions of the TTree zstd and RNTuple datasets:
  - root://eospublic.cern.ch//eos/root-eos/AGC/ttree-zstd/nanoAOD
  - root://eospublic.cern.ch//eos/root-eos/AGC/rntuple/nanoAOD
- 5/6 (AP): measures time evolution of CPU (%) and MB/s when reading an uncached AGC dataset
  - The gaps in mt come from RDF or the xrootd client



dask-local



multithreading

- 6/6 (AP): the new datasets are now on EOSPILOT:
  - /eos/pilot/rntuple/data-ec/v2/rntuple/
  - /eos/pilot/rntuple/data-ec/v2/ttree-zstd/
  - And the same under data-rep/v2
- 7/6 (AP): first implementation of JCache
- 13/6 (VP): new datasets condensed to have half the amount of files, with the RNTuple version generated with ApproxZippedClusterSize = 200 MB
  - ttree-zlib-condensed-2x
  - rntuple-zstd-condensed-2x
  - With a script to generate the input json file from the base dir (see MM)
  - AP reports that with condensed format times for (unached, cached) go from (120s, 110s) to (89s, 81s)

- 13/6 (AP): now JCache can be run either behind an XRootD proxy server or behind the client

AGC Dataset	Test Case	Realtime
RNTuple EC MT=64 DEMUX=64	EOSPILOT cold data	107s
RNTuple EC MT=64 DEMUX=64	EOSPILOT warm data	92s
RNTuple EC MT=64 DEMUX=64	JCache @ Proxy cold data - cold backend	127s
RNTuple EC MT=64 DEMUX=64	JCache @ Proxy cold data - warm backend	107s
RNTuple EC MT=64 DEMUX=64	JCache @ Proxy warm data	92s
RNTuple EC MT=64 DEMUX=64	JCache @ Client cold data	111s
RNTuple EC MT=64 DEMUX=64	JCache @ Client warm data warm BC	93s
RNTuple EC MT=64 DEMUX=64	JCache @ Client warm data cold BC	90s

- 17/6 (AP): compared I/O between the original and the condensed formats
- ROOT mt beats dask-local for the first time by 20 s in runtime!
- 1/7 (JB): new code fixes eliminate the “gaps” in CPU and IO for mt, by opening files in the background

```
RNTuple (standard)
# ------------------------------------------------------------ #
# JCache : total iops     read      : 5540
# JCache : total iops     readv     : 21588
# JCache : total iops     readvread : 212418
# ------------------------------------------------------------ #
# JCache : open files     read      : 923
# JCache : open unique f. read      : 796
# ------------------------------------------------------------ #
```

```
RNTuple (condensed)
# ------------------------------------------------------------------------ #
# JCache : total iops     read      : 14472
# JCache : total iops     readv     : 13224
# JCache : total iops     readvread : 68837
# ------------------------------------------------------------------------ #
# JCache : open files     read      : 693
# JCache : open unique f. read      : 405
# ------------------------------------------------------------------------ #
```

- 4/7 (AP): tested the latest changes, which give further improvements. Now a single client will run remote almost equally fast than with data cached on the local SSD! It now takes 82 s to read cold datasets remote
- 12/7 (AP):