XCache for analysis workflows

Nikolai Hartmann, Guenter Duckeck, Christoph Anton Mitterer, Rodney Walker

LMU Munich

September 22, 2020

What is XCache?

- Disk caching proxy using xrootd (`libXrdFileCache.so`)
- Data is cached in blocks
- Simply prepend xcache server url - e.g.
  `TFile::Open("root:[xcache-server]:[port]//[xrootd-path]")`
- Optionally use rucio DIDs via N2N plugin:
  https://github.com/xrootd/rucioN2N-for-Xcache
  → allows usage of rucio DIDs instead of xrootd path
  → tracks identical files distributed at different locations
  (internal symlink `.../scope/XX/YY/filename`)

Munich XCache Setup

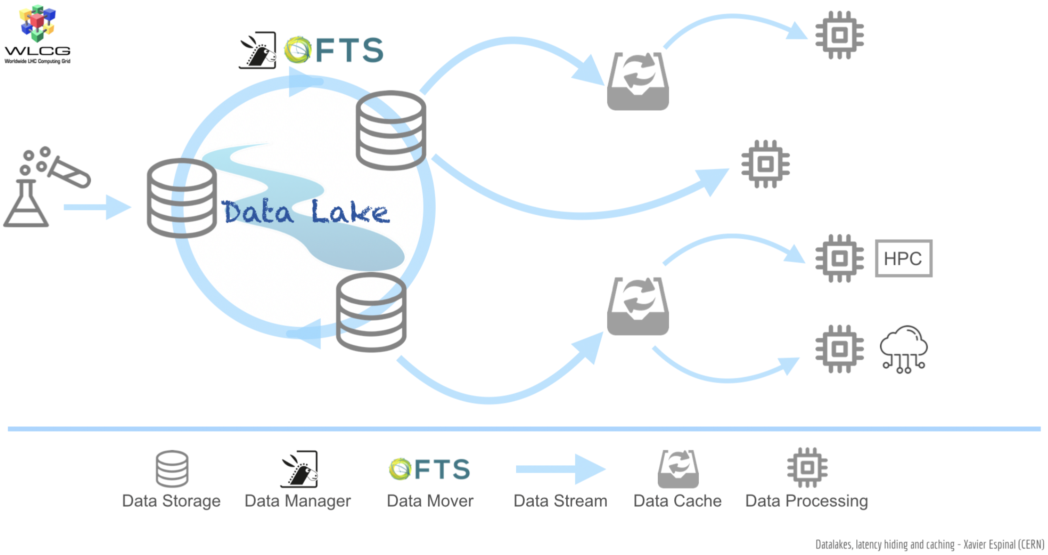
- Hardware: Old dCache pool node (from 2012):
  - Dell R710, 2x6 core Xeon L5640, 32 GB RAM, 10 Gb Ethernet
  - 60 TB Raid (2x12x3TB HDD)
    → now operated as individual disks
- Second node with similar Hardware for testing new versions/setups
  → also want to test "cluster" of XCaches
- Xrootd version 4.11.3
  → recently also started testing version 5
- Setup using singularity SL6 image. Full configuration:
  https://gitlab.physik.uni-muenchen.de/Nikolai.Hartmann/xcache-singularity-lrz/
- XCache settings:

```
pfc.ram 14g
pfc.blocksize 1M
pfc.prefetch 10
```

  → also experimenting without prefetch

The WLCG-data-lake model

Discussed within DOMA/ACCESS



(Graphic by Xavier Espinal)

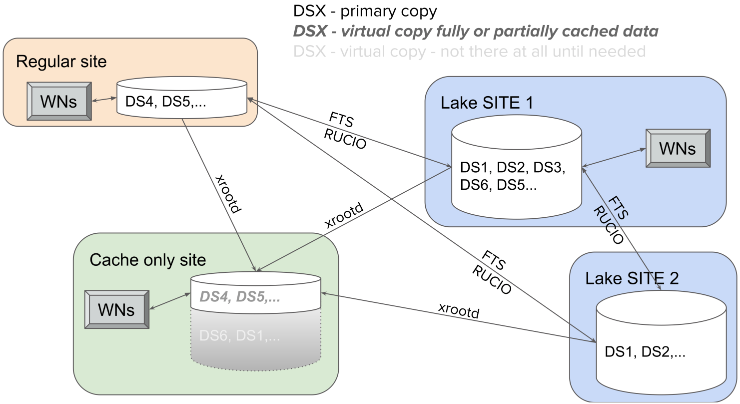
From production to analysis

So far we mainly studied accessing files for ATLAS production
(see slides from meeting last year)

Reasons to look more carefully into analysis workflows:

- Different access pattern (production jobs currently copy the whole file to scratch disk)
- More data re-use expected (especially with future analysis formats)
- Different access patterns when we consider columnar data analysis (we might do that in the future)

Virtual placement



(Graphic by Ilija Vukotic)

- “virtually” place datasets to cache-only sites
- expected to ensure high hit rates
- started including a queue that uses our Xcache server
- first results promising, but currently deactivated due to several issues (most of them now solved but take times to propagate fixes)

Bug: ROOT TChain with xcache paths

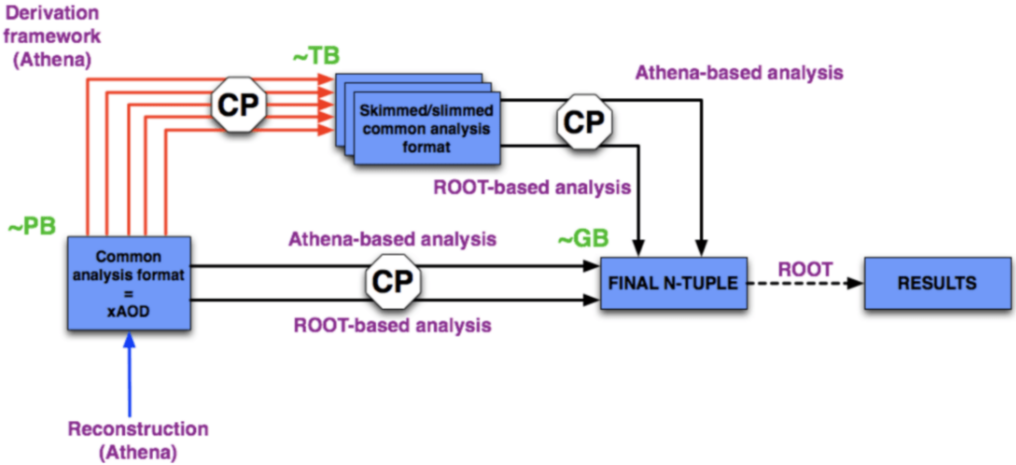
Already known (and fixed) since a while - ROOT bug parsing xcache paths with TChain:

```
root [0] TChain c
(TChain &) Name:  Title:
root [1] c.Add("root://lcg-lrz-xcache1.grid.lrz.de:1094//root://fax.mwt2.org:1094//pnfs/uchicago.edu/atlasscratchdisk/rucio/user/bngair/1c/26/1058501._000721.out
(int) 1
root [2] c.GetListOfFiles()->First()->GetTitle()
(const char *)
"root://lcg-lrz-xcache1.grid.lrz.de:1094//root://fax.mwt2.org:1094//pnfs/uchicago.edu/atlasscratchdisk/rucio/us"
```

- Fixed in ROOT 6.22, backported to 6.20, 6.18, 6.16
  → 6.16/02, 6.18/06, 6.20/06, 6.22/00 should have the fix (according to Philippe)
  → also see ROOT-10494 and the fix #4888
- Available in the lcg releases currently: 6.20/06, 6.22
- Takes time to propagate to analysis releases
  (TChain mainly used by analysis jobs)

The ATLAS plan for Run 3: change from this ...

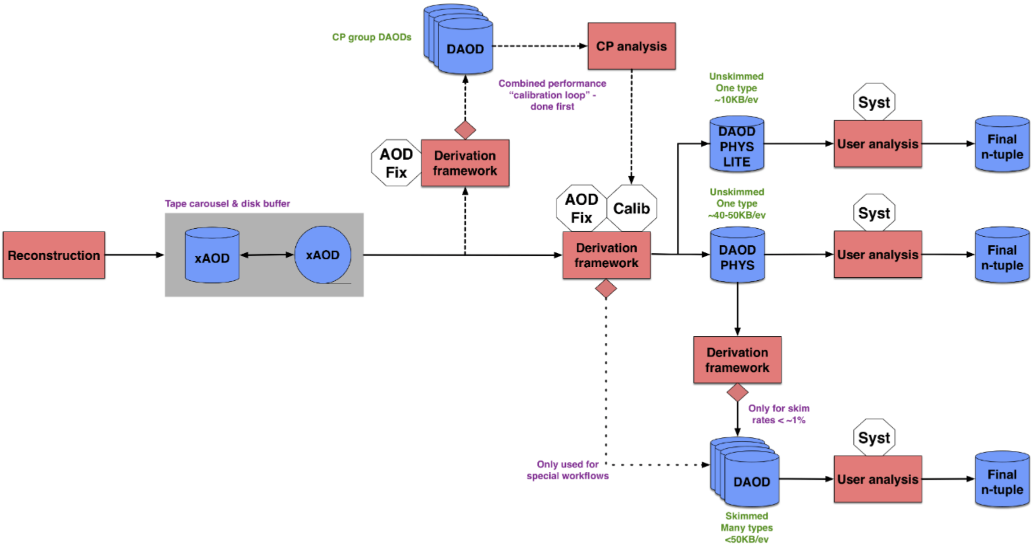
See also ATL-COM-SOFT-2019-027



Main problem: Too many DAOD (skimmed/slimmed) formats, takes too much disk space

... to this

See also ATL-COM-SOFT-2019-027



Columnar data analysis - Motivation

Operate on columns - "array-at-a-time" instead of "event-at-a-time"

Advantages:

- Operations can be predefined, no for loops! Most prominent example: numpy
  $\rightarrow$ this paradigm removes slow performing organisational stuff out of the event loop
- These operations run on contiguous blocks in memory and are therefore fast (vectorizable, good for CPU cache)
- Lots of advances in tools during the last years, since this kind of workflow is essential for data science/machine learning

Disadvatages

- Arrays need to be loaded into memory
  $\rightarrow$ need to process chunk-wise if amount of data too large
- Some operations more difficult to think about
  (e.g combinatorics, nested selections, variable length lists per event)

Columnar data analysis with ATLAS PHYSLITE

Idea:

- Most data is stored in "aux" branches (`vector<basic-c-type>`)
  → easily readable column-wise, also with uproot
- Reconstruction/Calibrations already applied
  → the rest might be "simple" enough to do with plain columnar operations
- Tools: uproot and awkward array

I/O ballpark estimate

Take current situation for ATLAS SUSY 1L analysis as an example:

- $\approx$ 200 TB of data with $\approx$ 50 kb/evt
- `DAOD_PHYSLITE`: $\approx$ 10 kb/evt
  $\rightarrow$ need to process $\approx$ 40 TB of data
  (probably a bit more since PHYSLITE unskimmed)
- Assume processing on local batch system: With 10 Gbit/s will take around 10 hours (with saturated network/150 kHz total)

Ways to improve this (get around network limit):

- Faster network connection
- Caching on the level of worker nodes
- "Intelligent data delivery" a la ServiceX
- Only read part of the data (few columns)
  $\rightarrow$ can be interesting e.g. for nominal only, early stages of an analysis
  $\rightarrow$ **study on the next slides**

I/O scaling tests

Some tests with a first PHYSLITE data sample (2015 data)

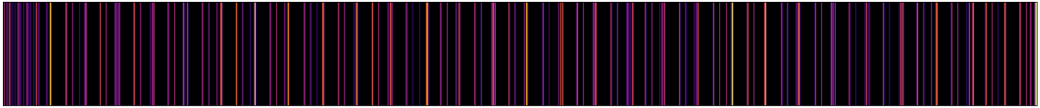
- read only few branches (same as in the 1L analysis test) of the $\approx$ 1 TB dataset
  $\rightarrow$ in theory $\approx$ **2% of the data**
- Read with $\approx$ **100 parallel tasks** (LMU batch system with dask)
- First test: read from LMU/LRZ dcache storage via xrootd (36 storage nodes)
  - Some issues with uproot (files not properly closed)
    $\rightarrow$ workaround, will be fixed in uproot4
    $\rightarrow$ some tuning of requested block sizes (will also not be nescessary anymore in uproot4, which supports xrootd vector reads)
  - After these fixes: data could be read within **5-10 minutes (7k files)**
  - Some storage nodes get rather busy with this access pattern
- Second test: read through xcache at LMU (1 storage node)
  - Gets extremely overloaded, not feasible anymore
  - Might become better when xcache is also extended to a cluster
  - But: maybe we can do better for this type of access (if we want to optimize for it)

Columnar data storage

- With current basket sizes in PHYSLITE these file accesses result in very scattered reading patterns
- A more columnar storage might help
- First try: store all "easily readable" branches in parquet files
  - Reading parquet via xrootd (using a small wrapper)
    (parquet files with 1 "row group" $\rightarrow$ one block per column)
  - Could read quickly ($\approx$ 3 min) even with single xcache node
  - Also good for block-wise caching (currently set to 1MB block size)
  - Not 100% fair comparison since only around 1/4 of data written to parquet
- Need to compare to very large basket sizes in ROOT
  $\rightarrow$ expect similar performance, but for the first test it was easier to produce the parquet files (awkward supports writing to arrow buffers)

Access patterns

Default DAOD_PHYSLITE



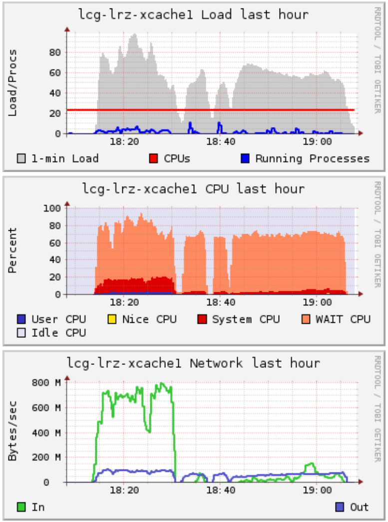
DAOD_PHYSLITE converted to parquet files (just "easily readable" branches)

DAOD_PHYSLITE with "jumbo baskets" (1 basket per branch)

(plotting details: histogram (128kiB bin width) of number of bytes read, clipped at a maximum of 1000)

Monitoring plots

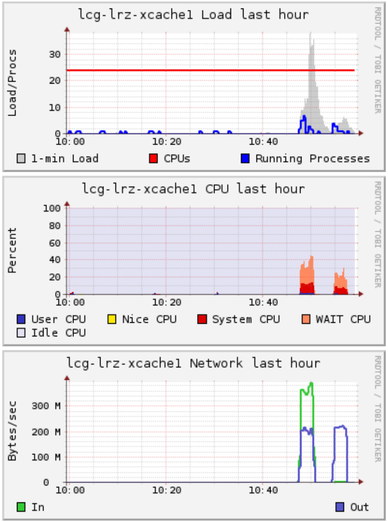
Reading default ROOT files



Load

I/O

Reading parquet files



Summary

- Analysis workflows are an interesting use case for caching
  $\rightarrow$ new data formats at ATLAS (and CMS) well suited
- In grid environment:
  Hit rate could be increased by "virtually" placing datasets to cache sites
- XCache could also help for columnar data analysis use cases
- Storage formats could be optimized for columnar access
  $\rightarrow$ would increase throughput for workflows that access only specific columns
  $\rightarrow$ integrates nicely with block-wise caching of large blocks

Backup

branches to read

```
#AnalysisElectronsAuxDyn.
electron_vars = [
    "pt",
    "eta",
    "phi",
    "DFCommonElectronsLHLooseBL",
    "DFCommonElectronsLHTight",
    "topoetcone20",
    "ptvarcone20_TightTTVA_pt1000",
]
#AnalysisMuonsAuxDyn.
muon_vars = [
    "pt",
    "eta",
    "phi",
    "DFCommonGoodMuon",
    "topoetcone20",
    "ptvarcone30",
]
#AnalysisJetsAuxDyn.
jet_vars = [
    "pt",
    "eta",
    "phi",
    "Jvt",
]
```