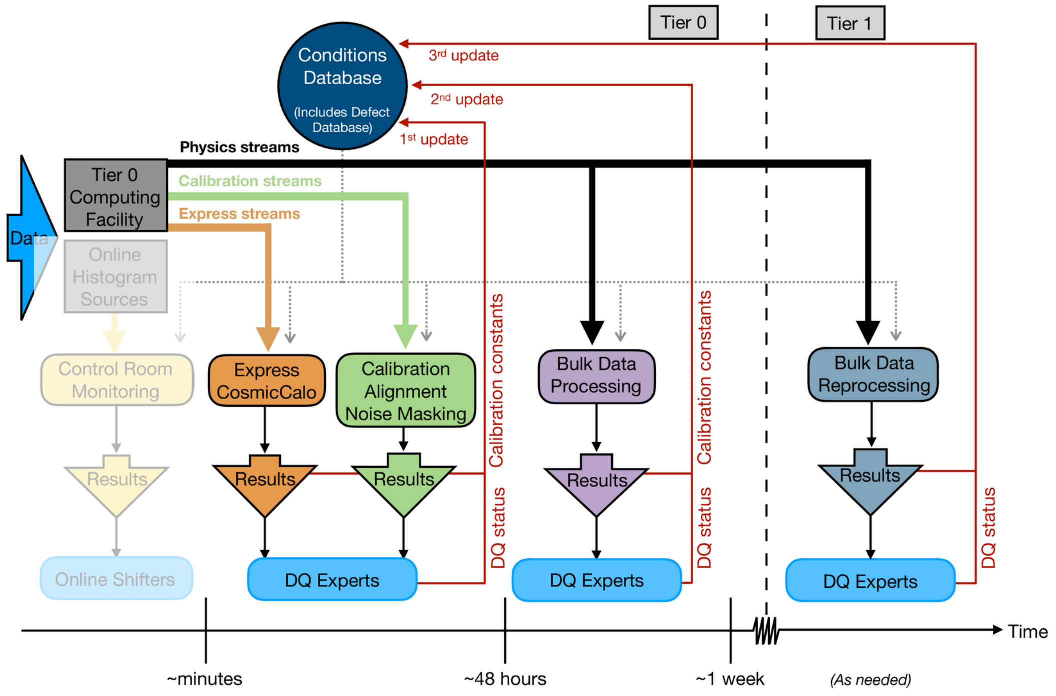
Recent Improvements to the ATLAS Offline Data Quality Monitoring System

Peter Onyisi

CHEP, 20 May 2021

Overview

- ATLAS offline data quality monitoring (DQM) chain is used to sign off data for good run lists and to provide feedback to optimize operation of the detector
- LHC shutdowns provide opportunity to update system



used in parallel with *online* chain which provides immediate (but limited) feedback in the control room; offline DQM allows comprehensive study of all recorded data

Projects

- Migration to multithreaded reconstruction
- New histogram postprocessing

$\left.\vphantom{\begin{matrix}a\\b\end{matrix}}\right\}$ this talk

- Versioned reference updates for automatic checks
  - improve responsiveness while keeping reproducibility
- Improved automatic check storage formats & disk layouts
  - halve file size with no difference in stored content by using better ROOT objects
  - reduce inode pressure by combining files in tar archives
- User interface improvements
  - dynamic interaction with plots in browser using JSROOT

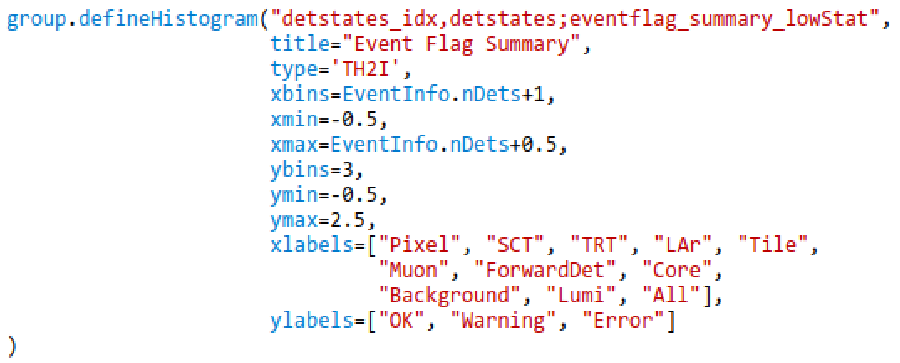
All discussed in accompanying paper; focus on first two here

Multithreaded Histogramming

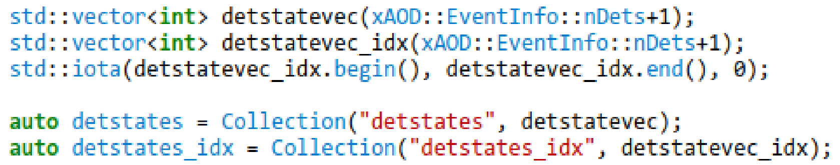
- Performance of detectors & calibrations is monitored with histograms created in the reconstruction workflow
  - O(10%) of data promptly processed in the "express" data stream, then full "Main" data 48 hours after end of run
- Reconstruction has been migrated to multithreaded "AthenaMT" in order to improve required memory/CPU core ratio
  - histograms need to be shared between threads to achieve good memory scaling
- MT-safe histogramming is tricky (and requires cooperation between all units): centralize hard parts
  - THistSvc design does not provide sufficient MT safety in histogram creation & management operations (MT-safe *filling* is "simple" part)
  - would like to avoid global ROOT locks
  - approach: programmers no longer touch raw histograms; all operations are handed in core libraries

Programmer Interface

- Histograms defined in Athena job Python configuration
  - extensive API to simplify tasks, e.g. for defining arrays of histograms for different detector regions
  - TH*, TProfile, TGraph, TEfficiency, TTree supported: can also support other backends although not used at present
  - histograms defined by variable names to be plotted
- Histograms filled in C++ event loop by providing variable names & data to histogramming tool
  - actual fill of histograms occurs in centralized code that determines what histograms can be filled given provided data



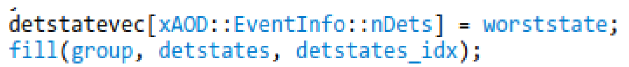
```
group.defineHistogram("detstates_idx,detstates;eventflag_summary_lowStat",
                      title="Event Flag Summary",
                      type='TH2I',
                      xbins=EventInfo.nDets+1,
                      xmin=-0.5,
                      xmax=EventInfo.nDets+0.5,
                      ybins=3,
                      ymin=-0.5,
                      ymax=2.5,
                      xlabels=["Pixel", "SCT", "TRT", "LAr", "Tile",
                               "Muon", "ForwardDet", "Core",
                               "Background", "Lumi", "All"],
                      ylabels=["OK", "Warning", "Error"]
)
```



```
std::vector<int> detstatevec(xAOD::EventInfo::nDets+1);
std::vector<int> detstatevec_idx(xAOD::EventInfo::nDets+1);
std::iota(detstatevec_idx.begin(), detstatevec_idx.end(), 0);

auto detstates = Collection("detstates", detstatevec);
auto detstates_idx = Collection("detstates_idx", detstatevec_idx);
```

// fill vectors



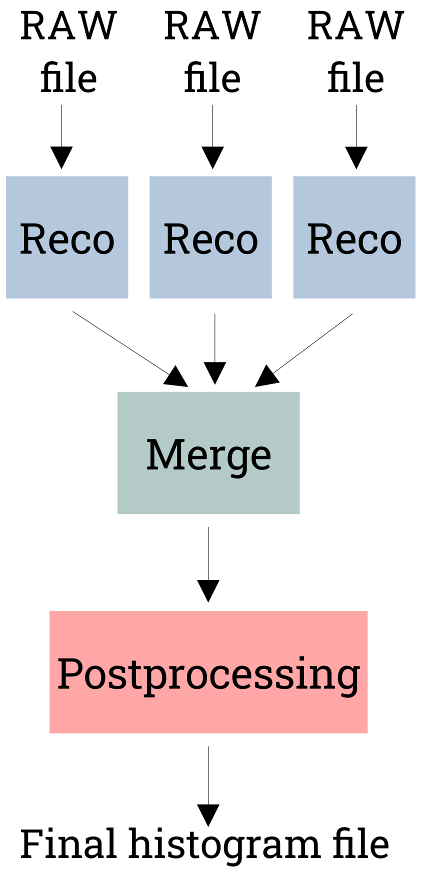
```
detstatevec[xAOD::EventInfo::nDets] = worststate;
fill(group, detstates, detstates_idx);
```

Lessons from MT Histogramming

- Design choice: every fill() call requires a (potentially expensive) lookup of what histograms to fill given provided variables
  - trigger and offline monitoring have different patterns → optimization of core code needs to handle both cases
  - encourage developers to use “vectorized” interfaces that fill many quantities at once, e.g. energy for all calorimeter cells in one call. “Cutmask” variables can be used to select a subset of a container for filling
  - new algorithms generally need a performance review to identify bottlenecks
- Try to minimize memory motion of data values
  - avoid intermediate representations of data when possible
- As always, lots of subtle places for race conditions to creep in
  - e.g. rebooking of time-dependent histograms in multiple threads

Histogram Postprocessing

- Histogram production has several phases
  - an accumulation step with commuting & reversible operations (usually addition to a bin of an array) in a single reconstruction job
  - merging of results from multiple jobs
  - optional "postprocessing" (e.g. make a new plot showing mean residuals)
- Similar to map-reduce (except that often reduce is "trivial" histogram addition)
- Postprocessed histograms are in general not possible to merge between jobs coherently
  - e.g. efficiency plots really need to keep numerator and denominator separate until final plot making, which is why ROOT now has TEfficiency
- In past, used to allow C++ postprocessing inside Athena
  - this isn't compatible with the new monitoring architecture (user algorithms have no access to the histogram data) so is now forbidden
  - introduce a new system to handle postprocessing

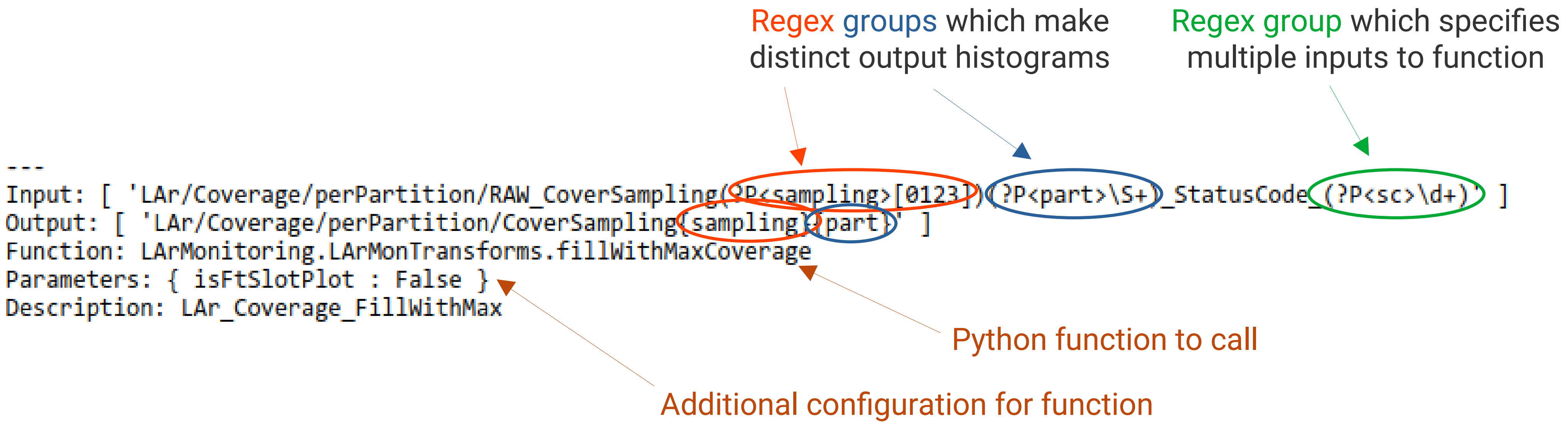


Postprocessing Engine

- Introduce generic framework for operations on histograms: histgrinder
  - complete factorization of histogram processing logic from access: handled via I/O plugins (ROOT file, ATLAS online histogramming system, Athena THistSvc, ...)
  - framework does not require any specific histogram technology
  - processing algorithms written in Python (but can use cppyy for speed)
  - pattern matching to simplify processing of similar histograms (e.g. different detector layers/regions)
  - for online operations: accepts histograms as they arrive & updates outputs
- ATLAS implementations:
  - offline: ROOT file → ROOT file
  - online: distributed online histogram (OH) system → OH
  - Athena piggyback for online: THistSvc → THistSvc in parallel to the reco job

Allows us to reuse postprocessing code in multiple environments

Example Histgrinder Configuration


Regex groups which make distinct output histograms
Regex group which specifies multiple inputs to function
Python function to call
Additional configuration for function

```
---
Input: [ 'LAr/Coverage/perPartition/RAW_CoverSampling(?P<sampling>[0123])(?P<part>\S+)_StatusCode_(?P<sc>\d+)' ]
Output: [ 'LAr/Coverage/perPartition/CoverSampling{sampling}{part}' ]
Function: LArMonitoring.LArMonTransforms.fillWithMaxCoverage
Parameters: { isFtSlotPlot : False }
Description: LAr_Coverage_FillWithMax
```

Same YAML configuration & user code used for offline and online applications; only difference is I/O plugins

Summary

- Multiple upgrades to ATLAS offline DQM chain in progress
- Focused on two developments:
  - multithread-safe histogramming in AthenaMT
  - new streaming histogram postprocessing framework
- Both deployed and ready for data taking