Multithreaded Data Quality Assessment in AthenaMT (+ consequences)

Peter Onyisi

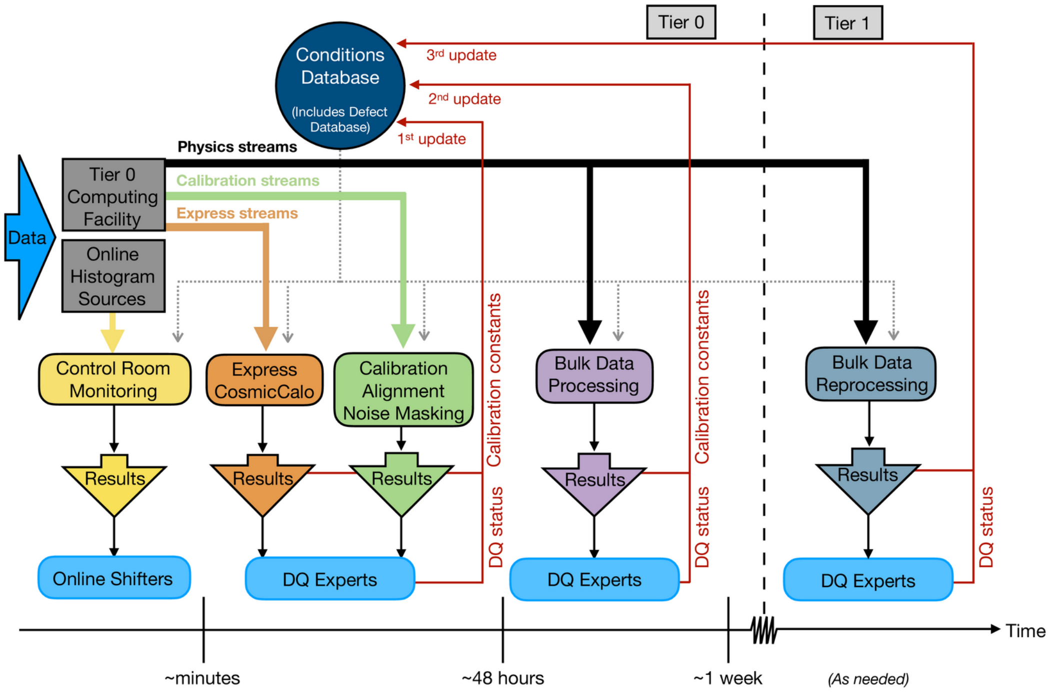
HSF Frameworks Meeting, 6 Oct 2021

TEXAS
The University of Texas at Austin

ATLAS EXPERIMENT

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
- Histograms produced in Athena reconstruction are a major input to the system
  - runs both offline and online
- LHC shutdowns + migration to multithreaded framework provide opportunity to update system



Introduction

- 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 after calibration loop
  - detector monitoring algorithms need access to RAW data or quantities only available during reco (not stored in AOD) – monitoring algorithms must coexist with reco
- Reconstruction has been migrated to multithreaded AthenaMT in order to improve required memory/CPU core ratio
  - earlier effort: multiprocess (copy-on-write) “AthenaMP”
  - DQ a major cause of large memory use of AthenaMP – each process gets its own copy of histograms
  - histograms need to be shared between threads to achieve good DQ memory scaling

Strategy

- Share a new jointly-developed core framework with High Level Trigger monitoring
  - able to leverage performance improvements, bug fixes, feature implementations from both
- 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)
  - avoid global ROOT locks as much as we can
  - approach: programmers no longer touch raw histograms; all operations are handed in core libraries
  - AthenaMT works fairly hard to make the other parts of algorithms re-entrant (execute methods are const, etc.)

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);
```

Critical Path: Lookup

- User code provides a list of what amounts to (variable name, variable data) pairs to framework in the fill() call
- Framework uses variable names to resolve which histograms need to be filled on each call
  - variable name scoped within a “group” (= Athena tool instance); can have many groups for a single monitoring algorithm
  - slightly pathological case: “a bunch of histograms for each of > 1000 muon detector chambers, distinguished by the name of the variables we call fill() with”
  - patterns quite different in offline and in trigger code
- User code usually calls fill() with a very small set of possible arguments, and set of histograms doesn’t change once job starts: get enormous speedup by caching the lookups
- Also provide methods for users to provide vector data instead of scalar data: only do expensive lookup once per event instead of inside tight inner loops
  - “cutmask” variables can be provided for histogram definitions to only plot a subset of a vector
- Performance checked with profiler runs with regularity
  - can be quite slow if naively used, but rarely a problem once code is optimized

Critical Path: Histogram Filling

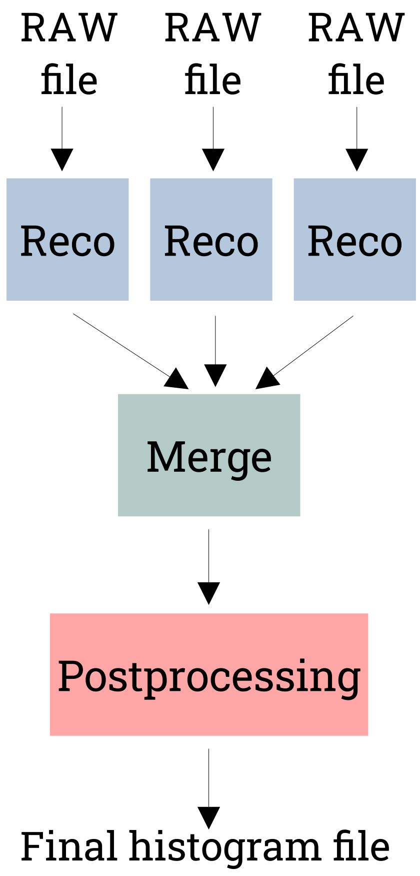
- Biggest issue: minimize memory motion of data values
  - avoid intermediate representations of data when possible
  - provide an interface to fill with arbitrary functions of elements of a collection (as long as collection supports operator[])
- As always, lots of subtle places for race conditions to creep in
  - e.g. rebooking of time-dependent histograms in multiple threads
  - THistSvc is missing an atomic "check if histogram exists, if not book" operation
- Actual filling protected behind per-histogram locks
  - Even running nothing but monitoring algorithms on an AOD, at 40 threads, DQ framework locks show no significant contention (much bigger effects in event I/O, etc.)

Histogram Framework Status

- At this point the framework is quite mature
  - still adding a few features as requested, e.g. “rolling last-N luminosity block” plots or prettier autogeneration of histogram titles
- Almost all Run 2 monitoring algorithms have been migrated to the new framework
- Strategy of isolating histogram-handling code in central, carefully-managed code has paid off
  - framework bugs can be fixed for all systems at once
  - clear best practices, things aren’t reinvented by every system
- DQ CI & other tests fairly frequently catch breakage in other domains (unintentional AOD output changes, subtle problems in multithreaded I/O …)
- Some small reduction in functionality largely addressed by next part of presentation…

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 arbitrary C++ postprocessing inside Athena (run at termination of a job)
  - 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

Identical Python postprocessing code in multiple environments

Example Histgrinder Configuration

Regex groups which make distinct output histograms

Regex group which specifies multiple inputs to function

```yaml
---
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
```

Python function to call

Additional configuration for function

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

Summary

- New histogram production framework introduced with multithreaded AthenaMT upgrade
- Necessitated parallel deployment of a new histogram postprocessing framework
- Both deployed and used for Run 2 data reprocessing validation & cosmics data taking