Data Production with DIANA in CUORE

Matteo Biassoni

Diana and Data Production Bootcamp - November 2020

Outline

- Data structure: RDCF, QRaw, Production files
- DAQ and analysis database
- Event concept and structure
- The Sequence
- The Module
- The online data production workflow
- The reprocessing workflow

CUORE

Data structure

In CUORE we have basically 3 types of data files:

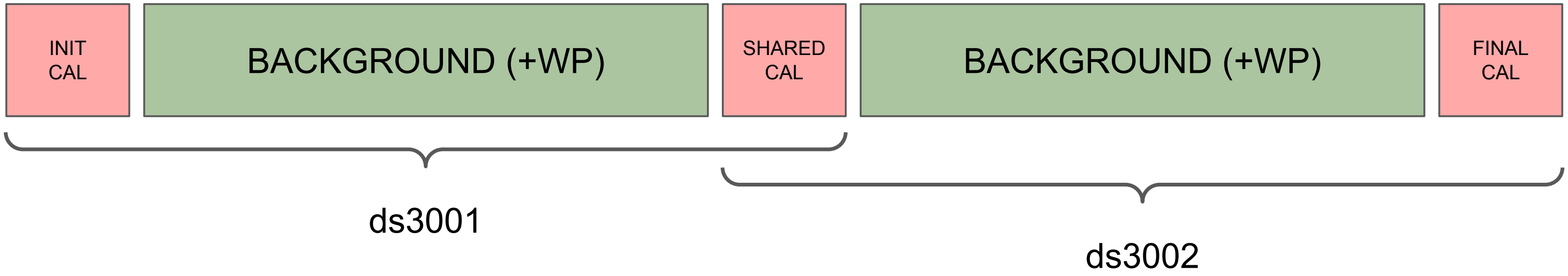
- RDCF
- QRaw
- Production (or any other name you choose)

All are rootfiles, can be opened and inspected directly from root, although contain some objects that need a working Diana installation to be understood.

All types of files are grouped by Run at all steps of the workflow.

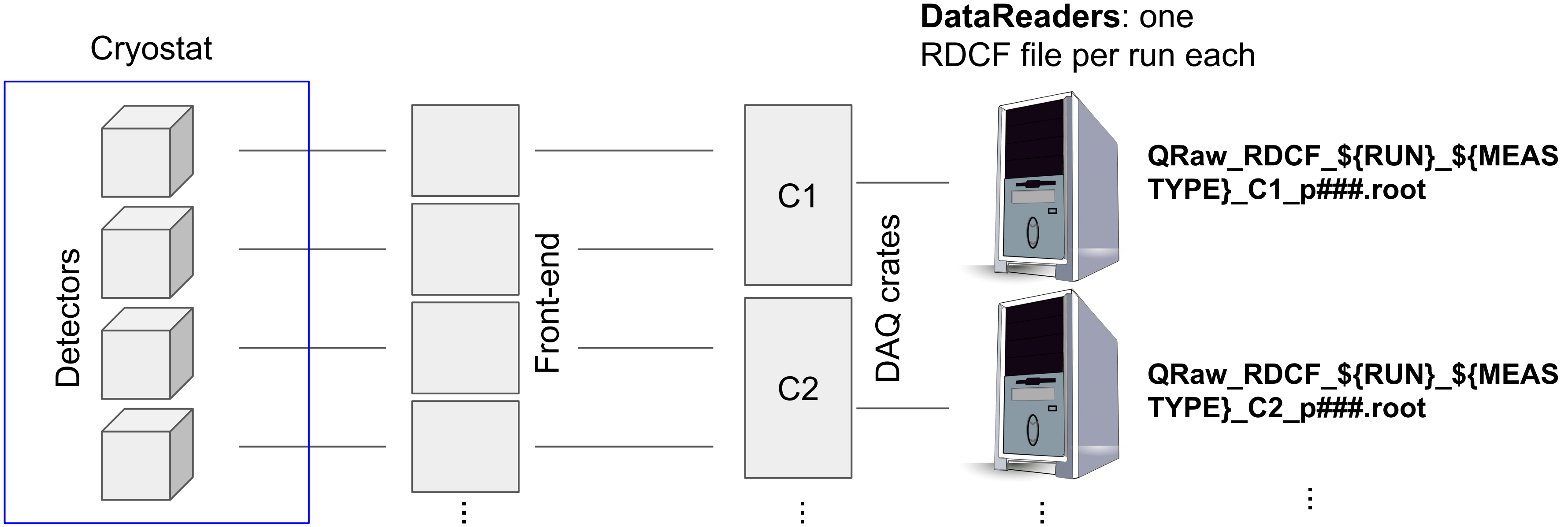
Data-taking organisation

- Data are collected during RUNs of ~24h duration
- Every ~1-1.5 months worth of data ~1week of calibration runs is acquired
- Initial calibrations + background + final calibrations form a DATASET
- Calibrations are typically shared among adjacent datasets
- Once a week a Working Point Measurement is taken



Data structure: RDCF files

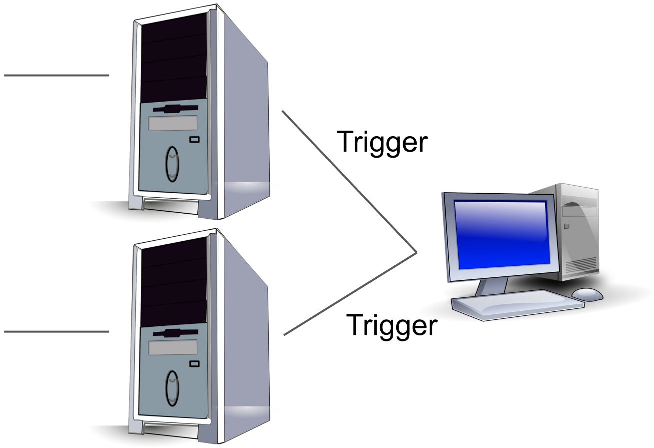
- Files that store the continuous data stream from the detectors
- Data are grouped by *DataReader*, no geometrical information
- Also list of triggers is stored



Data structure: QRaw files

- Files that store the information about triggered EVENT
- Data from all channels grouped in a single file
- Only basic DAQ-related information is stored, but the event structure is already created

DataReaders: one RDCF file per run each



DataReaders also run online trigger (signal, pulser, noise). Trigger information passed over to Builder

Builder: processes the trigger information from DataReaders, “builds” the events and store them in a single QRaw file per run

QRaw_${RUN}_${MEASTYPE}_p###.root

Data structure: Production files

- Created at the first step of the data production (DP from now on), called *preprocess*
- Can have any name, convention in use:

Production_${RUN}_${TOWER}_${MEASTYPE}_p###.root

MEASTYPE can be:

- C = Calibration
- B = Background
- T = Test
- R = Reprocess (with a different RUN that diana associates to the original via DB)

- Store the original events from QRaw files, grouped by RUN and TOWER (19 files per run)
- Each event is an entry of a QTree, a diana object based on root TTree
- More quantities are added to the event at any step of the DP
- Each step of the DP actually creates a new QTree which is a "friend" of the original one
  - each QTree can be stored in an independent file, but during "standard" DP all trees are "merged" in the same file and made friends

Data structure: Production files

- The high level content of Production files can be inspected with *diana-rootfilehandler*

```
biassonicuore@ui-tier1:/storage/gpfs_data/cuore/users/biassonicuore/CUORE_analysis/spring2019_reprocessing/output/ds3522> diana-rootfilehandler -l Pr
oduction_350084_001_R_p001.root
== Processing directory: /storage/gpfs_data/cuore/users/biassonicuore/CUORE_analysis/spring2019_reprocessing/output/ds3522/
Production_350084_001_R_p001.root:  QTrees { qtree_PileUp qtree_AveragePulse qtree_StabilizationDiscontinuities qtree_Amplitude qtree_GetAutoTrendVBo
l qtree_MultiStabilizationBaselineCorrection qtree_CalibrationCoefficientsHeaterTGS qtree_CalibrationCoefficientsCalibrationTGS qtree_MultiEnergy qtr
ee_EnergySelector qtree_ApplySelectedEnergy qtree_CoincidenceMultiplicityTower qtree_JitterByCoincidenceTower qtree_FastCoincidenceMultiplicity qtree
_ShapeCoefficients_LT qtree_ApplyShapeNormalizationLT qtree_FastCoincidenceMultiplicityAnalysis qtree_ApplyMahalanobisDistanceLT qtree_ShapeCoefficie
nts_HT qtree_ApplyShapeNormalizationHT qtree_GoodAnalyses qtree_ApplyMahalanobisDistanceHT qtree qtree_FastCoincidenceValidation }
```

- All qtrees are listed. Name is:
  - *qtree* for the first one (created by *preprocess* sequence)
  - *qtree_NameOfSequence* for the following ones
- *diana-rootfilehandler* can perform many operations on the files, including removing a specific tree. See *diana-rootfilehandler -h* for options

Data structure: Production files

CUORE

- From *root* we can inspect the content of a Production file with more detail:

```
biassonicuore@ui-tier1:/storage/gpfs_data/cuore/users/biassonicuore/CUORE_analysis/spring2019_reprocessing/output/ds3522> root -l Production_350084_0
01_R_p001.root
QStyles: Style"qprod" has been set
root [0]
Attaching file Production_350084_001_R_p001.root as _file0...
root [1] .ls
TFile**         Production_350084_001_R_p001.root
 TFile*         Production_350084_001_R_p001.root
  KEY: TDirectoryFile   Global;1        Diana global objects
  KEY: QTree    qtree_PileUp;1  pile-up flags
  KEY: QTree    qtree_AveragePulse;1    average pulse data
  KEY: QTree    qtree_StabilizationDiscontinuities;1    baseline stabilized data
  KEY: QTree    qtree_Amplitude;1       Amplitudes data with OF and wOF
  KEY: QTree    qtree_GetAutoTrendVBol;1        stabilization vbol data
  KEY: QTree    qtree_MultiStabilizationBaselineCorrection;1    baseline stabilized data
  KEY: QTree    qtree_CalibrationCoefficientsHeaterTGS;1        calibration data for heaterTGS stabilization
  KEY: QTree    qtree_CalibrationCoefficientsCalibrationTGS;1   calibration data for calibrationTGS stabilization
  KEY: QTree    qtree_MultiEnergy;1     energy calibrated data
  KEY: QTree    qtree_EnergySelector;1  energy selector data
  KEY: QTree    qtree_ApplySelectedEnergy;1     apply_energy selected data
  KEY: QTree    qtree_CoincidenceMultiplicityTower;1    coincidence information by tower
  KEY: QTree    qtree_JitterByCoincidenceTower;1        Jitter by coincidence
  KEY: QTree    qtree_FastCoincidenceMultiplicity;1     coincidence information
  KEY: QTree    qtree_ShapeCoefficients_LT;1    pulse shape coefficients vs. energy
  KEY: QTree    qtree_ApplyShapeNormalizationLT;1       apply pulse shape cut normalization
  KEY: QTree    qtree_FastCoincidenceMultiplicityAnalysis;1     coincidence information
  KEY: QTree    qtree_ApplyMahalanobisDistanceLT;1      data with mahalanobis distance, LT
  KEY: QTree    qtree_ShapeCoefficients_HT;1    pulse shape coefficients vs. energy
  KEY: QTree    qtree_ApplyShapeNormalizationHT;1       apply pulse shape cut normalization
  KEY: QTree    qtree_GoodAnalyses;1    good anlayses flags
  KEY: QTree    qtree_ApplyMahalanobisDistanceHT;1      data with mahalanobis distance, HT
  KEY: QTree    qtree;1 preprocessed data
  KEY: QTree    qtree_FastCoincidenceValidation;1       multiplet validation flags
```

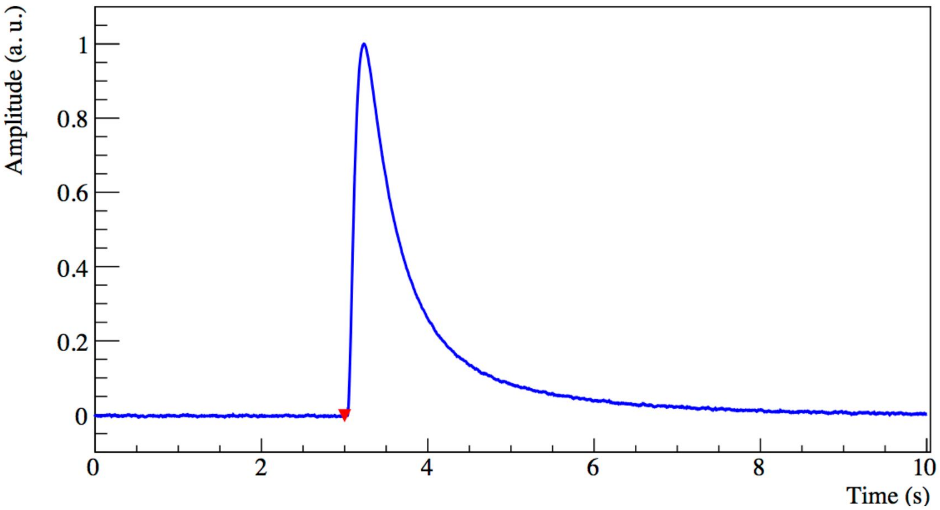
Global information, not event-specific

DAQ and Analysis Database

- Both DAQ and analysis rely on a psql database for some information:
  - run number and start-stop information
  - daq configurations (sampling frequency, channel mapping, channel to datareader)
  - continuous file bookkeeping (where are the RDCF files for a given run/channel)
  - run to dataset association
  - BadIntervals: sections of a run that should not be used during a given step of analysis
  - BadForAnalysis: run/channels that failed a given step of the DP and therefore miss some event quantities

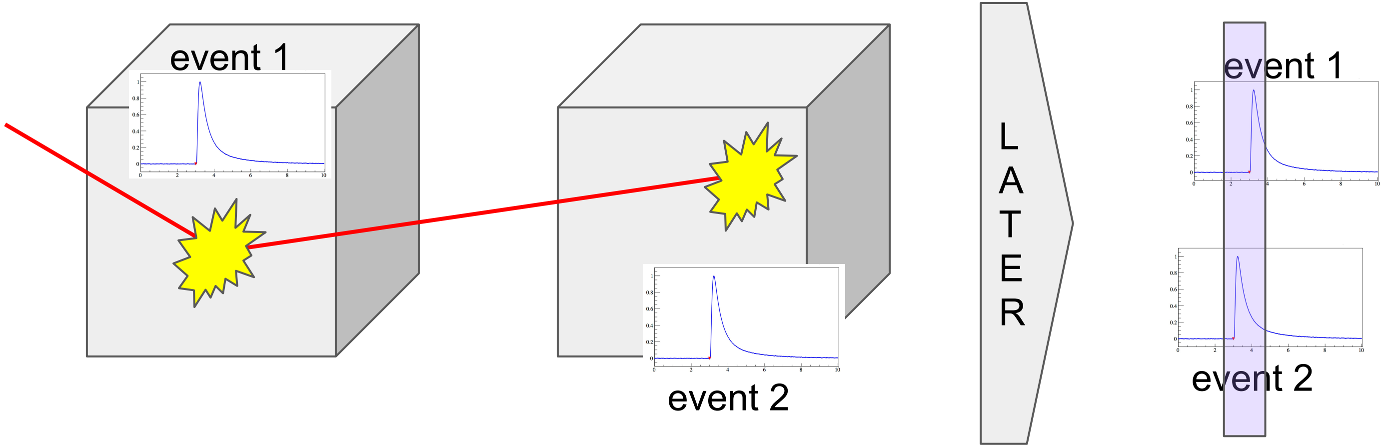
Event: concept

- The EVENT in diana is a collection of quantities associated to each trigger firing on a detector's channel
- The **trigger** can be generated either by a **signal** (derivative, OT, etc…), by a **pulser** injecting power on the crystal or randomly to record **noise** samples
- The signal triggers are supposed to fire as a consequence of a particle interacting with a crystal



Event: concept

- If a particle deposits energy in N crystals (e.g. compton scattering, surface alfa, muon), N independent signal events will be generated and stored
    - timing and geometrical correlations are reconstructed at a later stage



Multiplet info (delta T, coincident channel, etc…) are added to both events

Event: implementation

- Each event is an entry of a QTree
- The quantities associated to the event are stored as *branches* of the tree
- Branches typically contain diana objects:
    - some are re-implementation of standard data types (QBool, QBaseType, etc…)
    - some are more complex objects (QPulse, QPulseParameters, QOFData, etc…) with data members that contain the real information
    - see Stefano's talk on how to access this information and read and write QObjects

Event: example

```
root [2] qtree->Show(0)
======> EVENT:0
 DAQ@Header.      = (QHeader*)0x2a9bc50
 DAQ@Header.Cuore::QObject.fUniqueID = 0
 DAQ@Header.Cuore::QObject.fBits = 50364416
 DAQ@Header.fRun = 350084
 DAQ@Header.fEventNumber = 1069
 DAQ@Header.fTime.fUniqueID = 0
 DAQ@Header.fTime.fBits = 50331648
 DAQ@Header.fTime.fFromStartRunNs = 10048000000
 DAQ@Header.fTime.fStartRunUnix = 1501424830
 DAQ@Header.fIsThermalDetector = 1
 DAQ@Header.fIsMuonVeto = 0
 DAQ@Header.fIsApollo = 1
 DAQ@Pulse.       = (QPulse*)0x2f04ee0
 DAQ@Pulse.Cuore::QObject.fUniqueID = 0
 DAQ@Pulse.Cuore::QObject.fBits = 50364416
 DAQ@Pulse.fSamplesADC.fUniqueID = 0
 DAQ@Pulse.fSamplesADC.fBits = 50331648
 DAQ@Pulse.fFiller.fUniqueID = 0
 DAQ@Pulse.fFiller.fBits = 50364416
 DAQ@Pulse.fFiller.fRun = 350084
 DAQ@Pulse.fFiller.fChannel = 49
 DAQ@Pulse.fFiller.fStartT.fUniqueID = 0
 DAQ@Pulse.fFiller.fStartT.fBits = 50364416
 DAQ@Pulse.fFiller.fStartT.fValue = 7048000000
 DAQ@Pulse.fFiller.fStopT.fUniqueID = 0
 DAQ@Pulse.fFiller.fStopT.fBits = 50364416
 DAQ@Pulse.fFiller.fStopT.fValue = 17048000000
```

Branches starting with DAQ@ are those added to the event when it was originally created by the Builder. They exist already in the QRaw files.

Event: example

```
BCountPulses@CountPulsesData. = (QCountPulsesData*)0x1d95d40
BCountPulses@CountPulsesData.Cuore::QObject.fUniqueID = 0
BCountPulses@CountPulsesData.Cuore::QObject.fBits = 50364416
BCountPulses@CountPulsesData.fNumberOfPulses = 2
BCountPulses@CountPulsesData.fTimeIntervals.fUniqueID = 0
BCountPulses@CountPulsesData.fTimeIntervals.fBits = 50331648
BCountPulses@CountPulsesData.fTimeIntervals.fSize = 1
BCountPulses@CountPulsesData.fTimeIntervals.fStride = 1
BaselineModule@BaselineData. = (QBaselineData*)0x1d7a480
BaselineModule@BaselineData.Cuore::QObject.fUniqueID = 0
BaselineModule@BaselineData.Cuore::QObject.fBits = 50364416
BaselineModule@BaselineData.fBaseline = -2373.76
BaselineModule@BaselineData.fBaselineFlatRMS = 6.57814
BaselineModule@BaselineData.fBaselineIntercept = -2372.14
BaselineModule@BaselineData.fBaselineSlope = -0.00143996
BaselineModule@BaselineData.fBaselineRMS = 6.50984
BaselineModule_FullWindow@BaselineData. = (QBaselineData*)0x1d80310
BaselineModule_FullWindow@BaselineData.Cuore::QObject.fUniqueID = 0
BaselineModule_FullWindow@BaselineData.Cuore::QObject.fBits = 50364416
BaselineModule_FullWindow@BaselineData.fBaseline = -2349.73
BaselineModule_FullWindow@BaselineData.fBaselineFlatRMS = 50.8754
BaselineModule_FullWindow@BaselineData.fBaselineIntercept = -2354.6
BaselineModule_FullWindow@BaselineData.fBaselineSlope = 0.000972107
BaselineModule_FullWindow@BaselineData.fBaselineRMS = 50.7954
```

Quantities calculated during the data production are stored as branches with self-explaining names

Event: variable naming

```
BaselineModule@BaselineData. = (QBaselineData*)0x1d7a480
BaselineModule@BaselineData.Cuore::QObject.fUniqueID = 0
BaselineModule@BaselineData.Cuore::QObject.fBits = 50364416
BaselineModule@BaselineData.fBaseline = -2373.76
BaselineModule@BaselineData.fBaselineFlatRMS = 6.57814
BaselineModule@BaselineData.fBaselineIntercept = -2372.14
BaselineModule@BaselineData.fBaselineSlope = -0.00143996
BaselineModule@BaselineData.fBaselineRMS = 6.50984
```

Branch name:

```
Owner_ExtraLabel@Label. = (QObject*)Address
```

Name of the module that calculated the quantities stored in this branch (fixed). ExtraLabel is used if a module is run more then once (user-selected)

"Name" of the object where the quantities are stored (user-selected)

Class name

Memory address of the object

Event: variable naming

```
BaselineModule@BaselineData. = (QBaselineData*)0x1d7a480
BaselineModule@BaselineData.Cuore::QObject.fUniqueID = 0
BaselineModule@BaselineData.Cuore::QObject.fBits = 50364416
BaselineModule@BaselineData.fBaseline = -2373.76
BaselineModule@BaselineData.fBaselineFlatRMS = 6.57814
BaselineModule@BaselineData.fBaselineIntercept = -2372.14
BaselineModule@BaselineData.fBaselineSlope = -0.00143996
BaselineModule@BaselineData.fBaselineRMS = 6.50984
```

Variables are stored as data members (see Stefano's talk for more details)

```
Owner@Label.fDataMemberName = 12345
```

Can be either a public or private data member accessed via a dedicated Get/Set method

Event: variable naming

CUORE

```
RejectBadIntervals@Passed. = (Cuore::QBool*)0x589af50
RejectBadIntervals@Passed.Cuore::QObject.fUniqueID = 0
RejectBadIntervals@Passed.Cuore::QObject.fBits = 50364416
RejectBadIntervals@Passed.fValue = 1
```

```
RunDataLoader@Dataset. = (Cuore::QBaseType<int>*)0x6558e40
RunDataLoader@Dataset.Cuore::QObject.fUniqueID = 0
RunDataLoader@Dataset.Cuore::QObject.fBits = 50364416
RunDataLoader@Dataset.fValue = 3522
```

QBaseType are re-implementations of *int*, *double*, *float*, etc… and only have the fValue data member that contains the variable value

```
Owner@Label.fValue = 0.1e+2
```

QBool are a re-implementation of *bool*, Label is always *Passed*

```
Owner@Passed.fValue = 1/0
```

Module: concept

- A MODULE is the entity that performs operations on the event.
- Are usually linked into *sequences* (see later)
- The operation can be:
  - Read an event → Reader
  - Calculate some quantity to be added to the event, or to be used in a following step of the sequence → Module
  - Decide whether an event should undergo the following operations or not → Filter
  - Write the event with the new quantities added → Writer

Module: READER

- Read the event from an input file
- Its behavior is defined by the following list of instructions:

Common to all types of module

```
reader RootFileReader
verbosity = info
enable = true
InputFile = ${SCRATCH}/output/ds${DATASET}/Production_${RUN}_${TOWER}_${MEASTYPE}.list
DeleteInputFiles = false
EventNumberPrintStep = 10000
TimeProfilingOn = true
InclusiveMode = false
endmod
```

${VARIABLE} are passed from the command line with -V option

Whether all the *QTrees* already in the event should be loaded, useful if *AppendToInput = false* is selected in the Writer

Module: MODULE

- Calculates quantities based on:
  - other quantities already present in the event or calculated by previous modules
  - the waveform associated to a given event
- Can also perform operations (filtering) on the waveform itself, creating a filtered version of it
- The new event-specific quantities are stored in the event
- Can read (write) additional global (not event-specific) information from (to):
  - file (txt, root, etc…)
  - DB
- Usually needs some input parameters provided by the user

Module: MODULE example

- Its behavior is defined by a list of instructions like:

Name of the module

```
module FilteredPulseAmplitude
verbosity = debug
enable = true
CheckForValidSamples = false
FilteredPulseLabel = MCOptimalFilter@FilteredPulse
FilteredAPOwner = MCOptimalFilter
AvgPulseInput = ${SCRATCH}/avg/ds${DATASET}/average_pulses_ds${DATASET}_tower${TOWER}.root
NPSInput = ${SCRATCH}/avg/ds${DATASET}/average_noise_power_spectra_ds${DATASET}_tower${TOWER}.root
CalculateChiSquare = true
LookForPeak = 4
TimeProfilingOn = true
endmod
```

Module specific options, can be location of input/output files, booleans options, numerical values of parameters, etc...

Module: FILTER

- Decide whether an event should undergo the following steps of the DP
- Acts as a GATE for the single events
- The decision is taken based on quantities contained in the event
- “Thresholds” are typically passed via configuration or external files or DB
- The quantity calculated is a QBool that can be:
    - saved to the event (the result of the filter can be used by following sequences and high level analysis) (*WriteResult* option)
    - used by following modules and then ditched (*Save* option)

Module: FILTER

- Multiple modules can be combined with logical operations:
  - CASE resets all the previous filters
  - AND
  - OR
- One “special” filter (MFilterResult) is used to recover and reapply the result of a previously calculated filter (or combination of filters) that was saved (same sequence) or written to the event (different sequence)

Module: FILTER example

- Typical instructions for a filter:

```
filter FilterInInterval
enable = true
verbosity = info
Logic = AND
VariableLabel = ApplySelectedEnergy@SelectedEnergy
MinValue = 50
MaxValue = 10000
ExtraLabel = EnergyThreshold
Save = true
WriteResult = true
endmod
```

- `filter FilterInInterval`: Name of the filter
- `Logic = AND`: Logic operation with previous filters
- `VariableLabel`, `MinValue`, `MaxValue`: Filter specific options
- `ExtraLabel = EnergyThreshold`: Owner's extra label in case a filter is calculated more than once
- `Save = true`: Whether the filter result should be memorized within the sequence
- `WriteResult = true`: Whether the result should be written into the event

Module: WRITER

- Write new quantities into the output file
- Every reader creates a new QTree named `qtree_NameOfSequence` with branches corresponding to the objects created by the modules
- By default the new `qtree_NameOfSequence` is dumped to a new root file
- If *AppendToInput* option is set to true, the new `qtree_NameOfSequence` is written into the original input file and becomes a *friend* of the original `qtree`
- Can ditch from the output events that don't pass the last series of filters
- Defines the structure of the output file names

Module: WRITER

- NB: Aliases can be defined (and added to the qtrees):
  - shortcut to a given data member of a given object
  - simple name to be used instead of the full Owner@Label.fDataMember name when using `dianagui` (see Guido’s talk)
  - defined in a text file
- NB2: in principle a Writer is not always required in a sequence (the Reader is):
  - not needed if only “global” (non event-specific) information is calculated
  - not needed if output only to ancillary file or DB

Module: WRITER example

- Typical set of instructions for a Writer

```
writer RootFileWriter
verbosity = info
enable = true
OutputFilePrefix = CoincidenceFast
OutputFilesList = SyncWithTowerRun
FileIdentifier = SyncWithTowerRun
OutputDir = ${SCRATCH}/output/ds${DATASET}/
Description = coincidence information
AliasFileName = ${CUORE_INSTALL}/cfg/alias_spring2019.txt
TimeProfilingOn = true
SkipEvents = false
AppendToInput = true
endmod
```

File naming and description of the qtree

File with definition of aliases

Whether events that are filtered before getting to the writer should be removed from the output tree

Whether the output file should be merged with input and trees made friends

Sequence: concept

- A SEQUENCE is a collection of MODULES that need to be executed sequentially on each event
- Typically a sequence collects operations that are conceptually connected, for example all the modules and filters that are needed to build and apply a given filter to the waveform, to stabilize the gain, to calibrate, etc...
- The first module must always be a Reader
- The order of the operations is defined by a CONFIGURATION FILE (*.cfg)
- The config file collects the instructions previously described for the different types of modules
- A sequence can run on any subset of data:
    - run, dataset
    - tower, whole detector
    - any combination of the previous

Sequence: example

- Let's imagine a dummy sequence that should do the following (almost non-sense, but we don't care here):
    - select any event with:
        - only one pulse in each triggered window
        - which is not in a BadInterval
        - pulse amplitude between 0 and 1000
        - pulse highest point falls between sample 2500 and 3500
    - now apply the optimum filter but with different templates depending whether the signal is from particle (Signal) or from the pulser (Heater)

```
sequence MySillySequence

reader RootFileReader
enable = true
InputFile = ${SHARED_SCRATCH}/output/ds${DATASET}/Production_${RUN}_${TOWER}_${MEASTYPE}.list
DeleteInputFiles = false
EventNumberPrintStep = 10000
InclusiveMode = false
endmod

filter BadPulse
enable = true
Logic = CASE
NumberOfPeaks = 1
TimeProfilingOn = true
endmod

filter RejectBadIntervals
enable = true
Logic = AND
endmod

filter FilterInInterval
enable = true
Logic = AND
VariableLabel = PulseBasicParameters@MaxPosInWindow
MinValue = 2500
MaxValue = 3500
ExtraLabel = Position
Save = false
endmod

filter FilterInInterval
enable = true
Logic = AND
VariableLabel = PulseBasicParameters@MaxMinInWindow
MinValue = 0
MaxValue = 1000
ExtraLabel = Amplitude
Save = true
endmod

filter RawDataFilter
enable = true
Logic = AND
KeepHeater = false
KeepSignal = true
KeepNoise = false
KeepThermometers = false
WriteResult = true
ExtraLabel = Signal
endmod

module MCOptimalFilter
enable = true
AvgPulseInput = ${AVGDIR}/signal_AP_${DATASET}_${TOWER}.root
NPSInput = ${AVGDIR}/signal_ANPS_${DATASET}_${TOWER}.root
ExtraLabel = Signal
endmod

module FilteredPulseAmplitude
enable = true
CheckForValidSamples = false
FilteredPulseLabel = MCOptimalFilter_Signal@FilteredPulse
FilteredAPOwner = MCOptimalFilter_Signal
AvgPulseInput = ${AVGDIR}/signal_AP_${DATASET}_${TOWER}.root
NPSInput = ${AVGDIR}/signal_ANPS_${DATASET}_${TOWER}.root
CalculateChiSquare = true
LookForPeak = 4
ExtraLabel = Signal
endmod

filter FilterResult
enable = true
BoolLabel = FilterInInterval_Amplitude@Passed
Logic = CASE
endmod

filter RawDataFilter
enable = true
Logic = AND
KeepHeater = true
KeepSignal = false
KeepNoise = false
KeepThermometers = false
WriteResult = true
ExtraLabel = Heater
endmod

module MCOptimalFilter
enable = true
AvgPulseInput = ${AVGDIR}/heater_AP_${DATASET}_${TOWER}.root
NPSInput = ${AVGDIR}/heater_ANPS_${DATASET}_${TOWER}.root
ExtraLabel = Heater
endmod

module FilteredPulseAmplitude
enable = true
CheckForValidSamples = false
FilteredPulseLabel = MCOptimalFilter_Heater@FilteredPulse
FilteredAPOwner = MCOptimalFilter_Heater
AvgPulseInput = ${AVGDIR}/heater_AP_${DATASET}_${TOWER}.root
NPSInput = ${AVGDIR}/heater_ANPS_${DATASET}_${TOWER}.root
CalculateChiSquare = true
LookForPeak = 4
ExtraLabel = Heater
endmod

writer RootFileWriter
enable = true
OutputFilesList = SillyFiles_${RUN}_${TOWER}_${MEASTYPE}.list
OutputFilePrefix = SillyFiles
FileIdentifier = SyncWithTowerRun
OutputDir = ${OUTPUTPATH}
Description = silly description
AliasFileName = ${ALIASPATH}/aliasfile.txt
TimeProfilingOn = true
SkipEvents = false
AppendToInput = false
endmod

endseq
```

Sequence: example

- Let’s strip all the module-specific parameters from the configuration file

```
sequence MySillySequence

reader RootFileReader
DeleteInputFiles = false
InclusiveMode = false
endmod

filter BadPulse
Logic = CASE
endmod

filter RejectBadIntervals
Logic = AND
endmod

filter FilterInInterval
Logic = AND
ExtraLabel = Position
Save = false
endmod

filter FilterInInterval
Logic = AND
ExtraLabel = Amplitude
Save = true
endmod

filter RawDataFilter
Logic = AND
WriteResult = true
ExtraLabel = Signal
endmod

module MCOptimalFilter
ExtraLabel = Signal
endmod

module FilteredPulseAmplitude
ExtraLabel = Signal
endmod

filter FilterResult
BoolLabel = FilterInInterval_Amplitude@Passed
Logic = CASE
endmod
```

F1

F2

F3

F4

F5

F4*

F1 && F2 && F3 && F4

The OF is applied only to “good events” && “Signal”

Reset previous filters F5 and re-applies the result of F4, i.e. F1 && F2 && F3 && F4

```

filter RawDataFilter
Logic = AND
WriteResult = true
ExtraLabel = Heater
endmod

module MCOptimalFilter
ExtraLabel = Heater
endmod

module FilteredPulseAmplitude
ExtraLabel = Heater
endmod

writer RootFileWriter
enable = true
SkipEvents = false
AppendToInput = false
endmod

endseq
```

F6

The OF is applied only to “good events” && “Heater”

A new file is created with only the qtree from these sequence

All events are written to the file, also those that didn’t pass the last filter

Sequence: example

- Eventually we have:
  - OF with the signal template is applied to events passing F1 && F2 && F3 && F4 && F5
  - OF with heater template is applied to events passing F4* && F6
  - F5 and F6 are written into the event
- **IMPORTANT**: quantities (objects) calculated by modules are present also in events that don't pass the filters (e.g. noise events, events with multiple puses, amplitude > 1000, etc…), but these objects will have a VALIDITY FLAG set to FALSE

Sequence: example

- Once the sequence is defined, we have to submit a corresponding diana job.
- All the ${VARIABLE} in the config file must be passed from the CL
- For a complete list of diana options *diana -h*

```
diana -C cfg/mysequence.cfg -V VAR1 value1 -V VARN valueN
```

Official Data Production

- Official data production is performed in 2 steps:
  - online DP: minimal set of sequences to check data quality and give feedback to Detector Operation (noise anomalies, calibration compatibility, stabilization issues, etc…)
  - offline reprocessing:
    - retrigger all dataset with Optimum Trigger to lower the thresholds
    - run more sequences to calculate additional and more refined quantities:
      - multiple amplitude estimators
      - multiple gain stabilization methods
      - selection of best amplitude estimator
      - coincidences with delay synchronization → creation of multiplets of events
      - pulse shape analysis

Official Data Production: ONLINE

• preprocess_RUN_TOWER	basic quantities and filters
• average_pulses_DATASET_TOWER • average_noise_power_spectra_DATASET_TOWER	ingredients for Optimum Filter
• amplitude_RUN_TOWER	apply OT
• stabilization_discontinuities_RUN_TOWER • stabilization_baseline_correction_RUN_TOWER	stabilization of thermal gain
• calibration_heaterTGS_DATASET_TOWER • energy_RUN_TOWER	calculate calibration coefficients (initial calibration only) and apply
• blinding_RUN_TOWER	blind ROI

Official Data Production: REPROCESSING

- Offline reprocessing is performed at the end of each dataset, exploiting both initial and final calibration and all bkg runs to build the OF and calculate calibration coefficients
- Sequences tend to evolve when new algorithms are developed
- Latest version of the procedure (as well as previous ones) is detailed at

  http://wiki.wlab.yale.edu/cuore/SoftwareComputing/Spring2020_reprocessing

Q&A