Project ID 15591

Development of DQM software infrastructure: storing and reading the monitoring information from the histograms filled by online client applications into relational tables.

Adam Andrzejczak

Supervised by: Mr. Salvatore DI GUIDA, Dr. Federico DE GUIO

August 28, 2015

Abstract

In CMS the online DQM stores the monitoring information from several heterogeneous data sources into histograms, which are later sent to the DQMGUI for visualization. System for the handling of monitoring data is crucial for operating the detector and realizing whether or not it is undergoing failures: in particular, relational databases are the current best option for hosting such data. In this context a new DQM plugin DQMDatabaseWriter was developed, it provides interface which can be used in other DQM modules to drop desired data into the relational database. In addition, a python script provides possibility to read and visualize already saved records.

Storing the monitoring data

The online monitoring data is stored in the form of histograms by the DQM. It was desired to add a new functionality to this system - a possibility to store some of the histograms' data into the relational database for a further analysis and inspection. That is why, the DQMDatabaseWriter plugin was developed, which can be used by a module that inherits from DQMEDHarvester class. An appropriate configuration file has to be provided for such a module. An example is shown here:

```
import FWCore.ParameterSet.Config as cms
#module name:
DQMDatabaseHarvester        = cms.EDAnalyzer("DQMDatabaseHarvester",
#directory of the histograms:
histogramsPath              = cms.string("Physics/TopTest/"),
#luminosity based histograms:
histogramsPerLumi           = cms.vstring(      "Vertex_number",
                                                "Vertex_number", #repetitions are automatically discarded
                                                "pfMet",
                                                "NElectrons",
                                                "ElePt_leading_matched",
                                                "EleEta_leading_matched",
                                                "ElePhi_leading_matched",
                                                "ElePt_leading",
                                                "NJets",
                                                "JetPt_leading",
                                                "JetEta_leading",
                                                "JetPhi_leading",
                                                "NElectrons_HLT"
                                                ),
 histogramsPerRun           = cms.vstring(      "ElePt_leading",
                                                "EleEta_leading",
                                                 "ElePhi_leading"
                                                ),
#database configuration
DBParameters = cms.PSet(
authenticationPath = cms.untracked.string(''),
messageLevel = cms.untracked.int32(3),
enableConnectionSharing = cms.untracked.bool(True),
connectionTimeOut = cms.untracked.int32(60),
enableReadOnlySessionOnUpdateConnection = cms.untracked.bool(False),
connectionRetrialTimeOut = cms.untracked.int32(60),
connectionRetrialPeriod = cms.untracked.int32(10),
enablePoolAutomaticCleanUp = cms.untracked.bool(False),
),
connect = cms.string('sqlite_file:db1.db'),
)
```

The module used in this example is called DQMDatabaseHarvester, which is a derived class from a public base class DQMEDHarvester. Such class must include DQMDatabaseWriter header, which provides interface necessary to drop the data into the database:

```
//If the database does not exists yet, creates it.
  void initDatabase();

//Provides information about histograms to be treated as run based, must be called in dqmEndLuminosityBlock
  void dqmDbRunInitialize(std::vector < std::pair <MonitorElement *, HistogramValues> > & histograms);

//Drops run based histograms' data into the database
  void dqmDbRunDrop();

//Drops luminosity based histograms into the database
  void dqmDbLumiDrop(std::vector <MonitorElement *> & histograms, int luminosity, int run);
```

A working example that uses this interface is called DQMDatabaseHarvester.cc and is located in 'DQMServices\DatabaseIO\plugins' directory, which also contains DQMDatabaseWriter plugin itself. The configuration file required by cmsRun job is called DQMDatabaseHarvester_test_cfg.py and is stored in 'DQMServices\DatabaseIO\python\test' directory.

Reading the monitoring data

Once the database is supplied with records, one might want to read from it. In order to facilitate this process, a Python script query_database.py provides several functions. User can specify the desired run, luminosity and/or histogram name. SQL output is transformed by the script into a CSV format, which may be stored in the output file (if its name was provided by the user). All possible arguments are:

-h, --help	show this help message and exit
-d, --database,	path to the database
-u, --user,	the database username
-p, --password,	the database password
-r, --run,	specify run number for a SQL query
-l, --luminosity,	specify luminosity number for a SQL query
-n, --histogram_name,	specify histogram name for a SQL query, if you also specify run (but no luminosity), program will display plots of histogram properties vs luminosities
-s, --noplot,	suspends plot display
-f, --filename,	writes results to a file (terminal output suspended)

All arguments are optional, however in order to create a SQL query, it is necessary to provide either luminosity, run or histogram_name value.

Example of use:

```
python query_database.py -r 1 -n "Vertex_number" -s
INFO: Connected to the default database
QUERY: SELECT * FROM HISTOGRAM_VALUES WHERE RUN_NUMBER = 1 AND NAME = Vertex_number

NAME, PATH, RUN_NUMBER, LUMISECTION, ENTRIES, X_MEAN, X_MEAN_ERROR, X_RMS, X_RMS_ERROR, X_UNDERFLOW, X_OVERFLOW, Y_MEAN, Y_MEAN_ERROR, Y_RMS, Y_RMS_ERROR, Y_UNDERFLOW, Y_OVERFLOW, Z_MEAN, Z_MEAN_ERROR, Z_RMS, Z_RMS_ERROR, Z_UNDERFLOW, Z_OVERFLOW

"Vertex_number", "Physics/TopTest", 1, 12, 100.0, 1.17, 0.042555845661906455, 0.42555845661906455, 0.030091527046662177, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
"Vertex_number", "Physics/TopTest", 1, 14, 200.0, 1.17, 0.031709619991415874, 0.44844174649557356, 0.02242208732477868, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,0.0, 0.0, 0.0, 0.0, 0.0, 0.0
"Vertex_number", "Physics/TopTest", 1, 17, 300.0, 1.1533333333333333, 0.024708672941267384, 0.4279667692187744, 0.017471670190890725, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
"Vertex_number", "Physics/TopTest", 1, 29, 400.0, 1.1475, 0.02035581919255524, 0.40711638385110477, 0.01439373778766308, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
"Vertex_number", "Physics/TopTest", 1, 35, 500.0, 1.15, 0.01786057109949176, 0.3993745109543174, 0.012629330940315096, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
```

Without specifying “–s” switch, user may see a pop up window with graphical representation of histogram properties vs luminosities, which is presented in Figure 1.

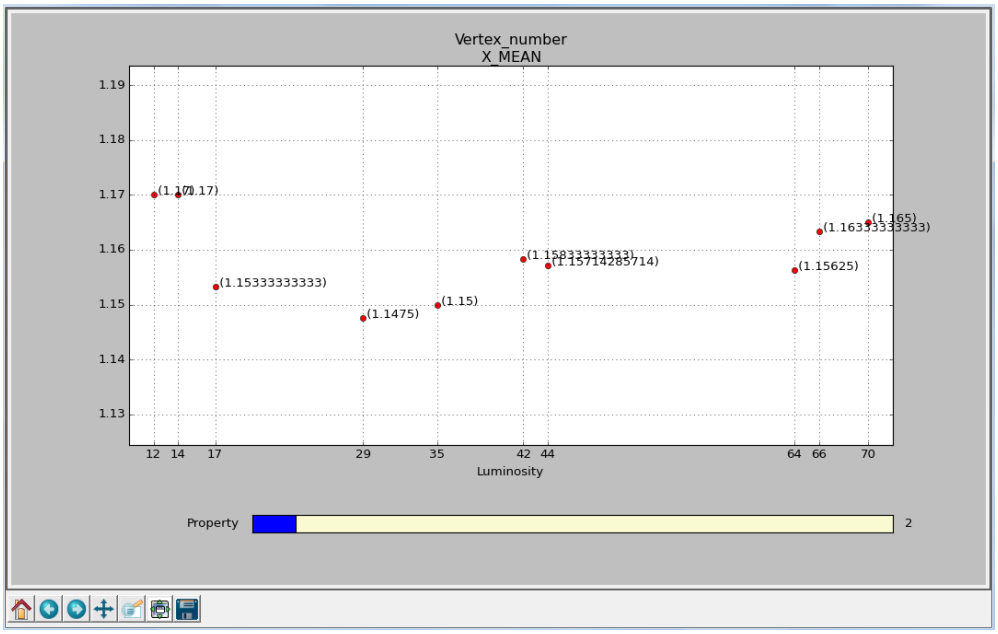

Figure 1 Graphical representation of histogram properties vs luminosities, user may use a slider “Property” in order to change between histogram’s properties

Conclusions

Over the course of 9 week project the DQM software infrastructure was developed and equipped with a new tool that allows experts to list histograms, which are the most vital, and to drop them into the database. The data can be easily accessed by the python script that converts SQL output into CSV format. Moreover, it is possible to notice the change of the histograms’ properties between subsequent luminosities, by looking at plots generated by the script, which can be useful for inspection by experts and shifters.