Generator Statistics and Simulation Tables

Manuel Barbera Asin
September 2006

Index

1. Introduction

The purpose of this project is to build a script which was able to generate statistics over several jobs and show them in html tables. Essentially, it should be made, each time, for one Event Type and one Gauss Version.

For each Event Type several productions are created, each time with a different configuration. Each of these productions are referenced by only one production identifier (prodID).

This configuration contains a `workflow`, the `workflow version`, a `gauss version`, the `decfiles version`, `paramfiles version`,...

A lot of jobs are generated for each production, and these jobs contains the useful information for our purpose.

So, the goal is to can generate statistics for one of each configuration. It doesn't matter the production, it will be done for the productions available. If one production has not enough jobs to make a good statistic, it will be completed with other production with the same configuration (if there are more).

There is also a possibility to generate the statistics for a given production, or a given workflow.

The script should can to be called to make two types of statistics: the own statistics and an average of the simulation tables. Both have to work with the same log files, so we can also ask the script for generate both at once.

2. How to use it

The script takes two mandatory arguments and several ones optional.

The usage man page is:

```
python statistics.py -e EVENTTYPE -g GAUSSVERSION
                     [--simulation/--generation]
                     [--publish]
                     [-p prodID][-l luminosity]
                     [-h/--help]
```

The -e argument takes the value of the Event Type for which we want to generate the statistics. That value must to be an 8 digits number.

The -g argument takes the value of the Gauss Version. For one gauss version we could have several productions with the same Event Type.

The arguments `--simulation` and `--generation`, tell the script the kind of statistics to generate. By default, the script will be executed for both processes.

`-p` takes a production identifier. The script will run only for this production, even if there are not enough jobs to make a good statistic. This value musts to be an 8 digits number.

`-l` option takes the workflow version or luminosity. It could be we want to generate statisctics only for one workflow version productions and not for the others.

`-h` and `--help` will show the usage page without execute the script.

the option `--publish` is to validate the results. Normally, each user will make his tests in his own account. If he wants to validate them, and make them available to everyone, the argument `--publish` have to be given. That will update the results in the

server.

Examples:

```
$ python statistics.py -e 11154100 -g v25r2 --publish
$ python statistics.py -e 13264400 -g v25r0 -l v1-lumi2 --simulation
$ python statistics.py -e 11144100 -g v25r1 -p 00001330 --generation
$ python statistics.py -e 10000000 -g v24r8 --generation --publish
$ python statistics.py --help
```

3. Program explanation

The script is written in python to make it compatible with windows and linux systems.

The useful information can be found in Gauss logfiles stored on the storage web side accessible from the web:

```
path = http://lhcb-logs.cern.ch/storage/lhcb/production
```

There is a configuration, `config_file`, file where this information, as many other, is keeped. The goal for this file is to keep some variable information there: storage path, the name of the results files, the directories where the results will be stored, the number of needed jobs to have a good statistic,... These data could change in the future and it's easier to change just some lines in the configuration file than to modify the whole script.

In the storage web site we can find all workflows (`DC04, Higgs, RTTC, DC06`) which have different versions (`v1, v2, v2r3, v1-lumi2`, etc...).

For each workflow version there will be several productions referenced by a production identifier (`prodID`). These identifiers are always 8 digits and they contain several jobs referenced with a job identifier (`jobID`), also with 8 digits. All logfiles produced by these jobs are stored in a `LOG/<jobID>` directory.

So a job is found in:

```
<path>/<workflow>/<version>/<prodID>/LOG/<jobID>
```

A jobs contains usually 3 Gauss logfiles, 2 Boole logfiles and 1 Brunel logfile:

- The absence of Brunel logfile means that the job crashes.

- Gauss step 1, called `Gauss_<prodID>_<jobID>_1.txt.gz`, has produced the selected event type for the production.
- Gauss step 2 and 3 produced minbias events.

Gauss logfiles belonging to the same production (at the same time they are associated to the same event type) should be read in to retrieve some information which will be sum up over several jobs to give a result per event type.

There is a minimum number of jobs required to get a good result. This value change between some event types, but for the majority it is the same.

Example:

`http://lhcb-logs.cern.ch/storage/lhcb/production/DC04/v2r3/00001045/LOG/00000001/`

Log files for Run 00001045_00000001

```
job.info
00001045_00000001.xml.gz
153922_parameters.txt.gz
job.output.gz
Gauss_00001045_00000001_1.txt.gz
00001045_00000001_Catalog_pool.xml.gz
gauss_monitor_00001045_00000001_1.hbook.gz
bookkeeping_00001045_00000001_1.xml.gz
Gauss_00001045_00000001_2.txt.gz
gauss_monitor_00001045_00000001_2.hbook.gz
bookkeeping_00001045_00000001_2.xml.gz
Gauss_00001045_00000001_3.txt.gz
gauss_monitor_00001045_00000001_3.hbook.gz
bookkeeping_00001045_00000001_3.xml.gz
Boole_00001045_00000001_4.txt.gz
boole_monitor_00001045_00000001_4.hbook.gz
bookkeeping_00001045_00000001_4.xml.gz
Brunel_00001045_00000001_5.txt.gz
brunel_monitor_00001045_00000001_5.root.gz
bookkeeping_00001045_00000001_5.xml.gz
```

The information in the Gauss log files is which is used by the script to calculate the statistics and create the tables.

These tables will be stored in html files. There will be a file for each process, workflow version and gauss version, which will contain all the statistics made for the productions with the corresponding configuration.

For example, there will be a file `Simulation_DC06_v1-lumi2_v25r0.html` which will contains the simulation tables for the productions generated with the v25r0 gauss version and for the `DC06/v1-lumi2`. Idem for `Generation_DC06_v1-lumi5_v25r2.html`.

4. Program Code

The script must search information in several files whether it be from a url location (Gauss and Brunel .txt.gz logfiles) or a local configuration file for apply several actions depending on it.

To make it possible the script uses several python libraries: `string, sys, os, urllib, gzip`. We need them because the script works with strings, system calls, interacts with the operating system, with url locations and gzip files.

The script works also with the bookkeeping database, an Oracle database. For this reason its necessary another library, called `cx_Oracle`, installed in the system.

First, the script checks the parameters passed. There is a python library, `getopts`, very useful for this purpose. Anyway, the defined function options take the values of the parameters and put them in variables.

Then, the script must take the configuration information from the configuration file. Information like the url path, the name of the results files, the number minimum of jobs needed, the directories where store the results.

If the parameter `--publish` is not passed, the `LocationPath`, where results will be putted, has to be changed for the local path.

It also takes a look along the productions files for the given gauss version looking for the Event Type (there could be several productions for the same Event Type), its description, the workflow version (its programmed for DC06 workflow but it can be easily updated), gauss version and the prodID values.

If no production is found, it could mean that the productions file has to be updated, so the script offer to consult the `bookkeeping` database to get the new values (if there are any). It will call the script `getproductions.py`.

If even after that, no production the script stops and finishes.

Normally, it should find some productions with the given configurations. The function productions give us also the different workflow version or luminosities availables, because the script should count on all of them and run once for each.

The different king of executions, simulation and generation, have also to be controlled. By default, the script will run for both, to take advantage to take just once the information from each log file, but it's possible to make run just for one of them. This fact is controlled by a variable, executions, which includes the process/es to do.

At this point, the script execute the main curl.

While there are productions with different configuration, it should be running to complete the statistics for all of them, and also if there are some productions without enough jobs.

Almost all the operations to do are the same for the simulation and the generation process, but there are some ones exclusive for each process. For example, for the simulation process it's necessary to know the `xmldddb` version, and it has to be taken from the database.

Before start with each process, the script checks if there are already some results for the current configuration, because the process will overwrite the previous results with the new ones.

Then the script goes on building the path where the log files should be found (with the production data recovered before), and looking for all the jobs available in the web site counting them to see if there are enough or not. This is a little rustic method to do it, and if ever the storage change, maybe it doesn't work anymore. The lines containing `<img src="/icons/layout.gif" alt="[DIR]" />` are the lines for the valid jobs. There are some other lines associated to other icon which don't have the log files needed.

At least none jobs was found the curl to cover the jobs begins. This involve the

script to for each job available, while it doesn't exceed the minimum jobs needed, to check if the Brunel and Gauss files are there. The Brunel files, for us, are just to check that the job is fine and didn't crash. There is a function charged to check it, and is called testpath.

```
<path>/<workflow>/<version>/<prodID>/LOG/<jobID>
```

The function take_info will recover the data kept in each file, uncompressing the txt file inside each `Gauss_<prodID>_<jobID>_1.txt.gz` file. Then the script will work always with this information.

It should check if the Event Type value in each log file fit with the Event Type given. If it does not fit, it should continue with another log file, because the statistics will be created for Event Type.

Next, for each process, generation or simulation, the corresponding lines will be recovered and put the values in several variables. This variables are dictionaries, whose will allow us to recover the values easier.

The lines to search and the operations to do with for the generation process can be seen in the web site below:

```
https://uimon.cern.ch/twiki/bin/view/LHCb/GengaussCounters
```

(For the simulation process, they are in a pdf file Simulation_tables.pdf which can be seen where this file is stored)

If the retrieved results are correct, which means they have a value, not 0, the next thing to control is if the current process need to be completed with another production or not. The script checks if there are enough jobs. If more jobs are needed, it will search another production with the same configuration. If some other is found, the curl above will be executed again for the new configuration adding the results to the previous ones.

Once, we have a good results, or if there are not more productions to complete a process without enough jobs, the results have to be stored in html tables.

There are different functions for each process (`create_tables` for simulation, and `calculations` and `store_results` for generation).

The functions calculations will build a string making the corresponding operations. In the beginning it was to store it in a txt file, so this operations could be changed and included in the store_results function.

This string will be passed to the store_results function to create the tables.

The function `create_tables` will make the same but directly, without creating a string with a txt file format.

Then it has to be checked if there is still more work to do (if the results are correct, if we have still more workflows versions to work with). If yes, the curl continues, and if not, it's over. It just has to send an email with a list of the `G4 with no PDGEncoding` found in the log files (for the simulation process).

5. getproductions.py code

The script makes a query to the database to recover the production informations needed for the main script. The `cx_Oracle` pythons library is needed.

It just takes a gauss version as argument and makes a query to the database to recover all the productions generated for.

The script makes a file, called `Productions_GAUSSVERSION.txt`, which will be stored in the directory productions and which lines are like:

```
#EventType Description WorkflowVersion ProductionIdentifier GaussVersion
```

Usage

```
$ python getproductions.py -g GAUSSVERSION
```

Example:

```
#EventType Description Workflow ProdID GaussVersion
10000000 incl_b v1-lumi5 00001325 v25r0
11154100 Bd_JpsiKS,ee v1-lumi2 00001329 v25r0
11144100 Bd_JpsiKS,mm v1-lumi2 00001330 v25r0
11102200 Bd_Kstgamma v1-lumi2 00001331 v25r0
```