2018/2019

Fermilab Summer School 2018

Final Report

# Black Holes Detection in Fermilab's Global Computing Grid

**Intern:**

**Lorenzo Lamberti**

**Supervisor:**

**Kevin Retzke**

## Abstract

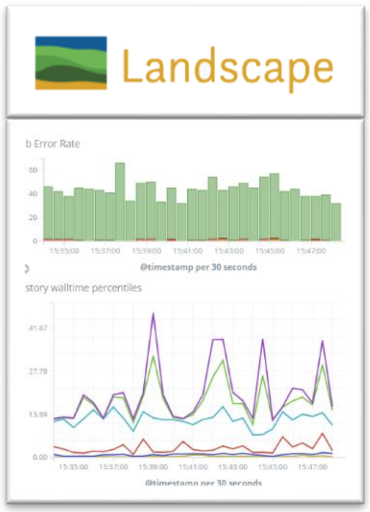
There are a lot of great challenges for utilizing widespread distributed computing resources (for example Open Science Grid, Cloud computing and High Performance Computing) through a scheduling/queueing system.
One of the main issues is the detection of faulty nodes of the grid, that are called ***blackholes,*** because they cause failure of the jobs submitted to the distributed system. It's important to detect these nodes as soon as possible to not waste computing resources and to minimize the number of failed jobs that are not due to user errors.
The aim of this work is to develop algorithms to analyze monitoring data streams from Fermilab's global computing grid in order to rapidly detect and identify aberrant conditions.

# Summary

# 1. Working Environment

The data of Fermilab's experiments are processed through Fermilab's **Global Computing Grid.** The term "Grid Computing" indicates the usage of distributed computing resources that work through a scheduling/queueing system. This system in particular is widespread all over the world.

**FIFE** (Fabrique For Frontier Experiments) is a high-level project that provides offline computing services for all Fermilab's experiments.
**Landscape** is a project born with the intention to create a monitoring system for FIFE and other scientific computing services at Fermilab.


Landscape
b Error Rate
@timestamp per 30 seconds
story walltime percentiles
@timestamp per 30 seconds

There are a lot of data to analyze in this complex system and some graphic tools are very useful to gather information. **Grafana** and **Kibana** software are used for the visualization of collected data from FIFE:

- Grafana is used for a general overview
- Kibana is useful for a more in-depth look

Both Kibana and Grafana work on **Elastisearch's server**: it stores detailed documents and logs about the jobs submitted. Its data storage engine is Apache Lucene. The data on this server is organized in indices and fields.

All the algorithms implemented are written through the **Python** programming language.

## 2. Training required

All the environments listed before require a proper training to better understand all the tools available for the analysis of data. If none of these tools are already familiar to the user, this should be the chronological path to be followed:

1. Learn Python programming language**:**
    1. Google's online python class;
    2. Python programming course by Marco Mambelli.
2. Understanding Landscape's monitoring tools**:**
    - Elasticsearch database;
    - Grafana software;
    - Kibana software.
3. FIFE's data exploration through Grafana and Kibana
    - Understanding data structure;
    - Learn **JSON queries** language;

# 3. Challenges

In order to get familiar with all the tools available for the analysis and explore FIFE's monitoring data it was set up a list of challenges to be accomplished. This chapter is going through every challenge followed by the answer to the question, the source of the data and code written (if the graphic tools Kibana and Grafana were not sufficient to answer to the question).

## 3.1 Challenge 1

1. **How many jobs were run in the past week?**

   Source of the data: fifebatch-history-*

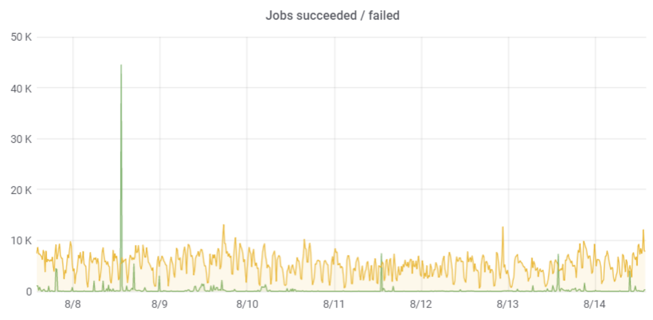
   It was just counted how many jobs were present in this index of memory. The output is a number (no graphs).

2. **How many succeeded (exit code 0) and how many failed?**

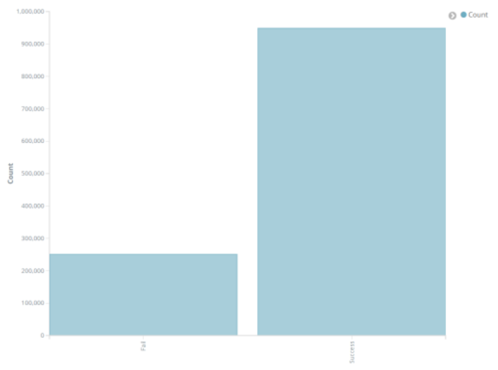
   Source of the data: fifebatch-history-*

   It was put a double filter for successful jobs (Exit Code=0) and failed jobs (Exit Code!=0).

   The result is displayed with both Grafana (count over time) and Kibana (total count)



**Grafana**



**Kibana**

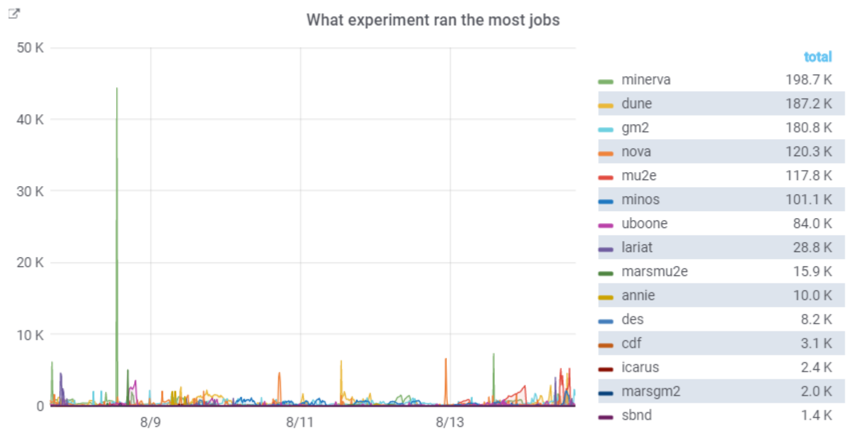
3. **What experiment ran the most jobs?**

   Source of the data: fifebatch-history-*

   Aggregation by: Jobsub_Group

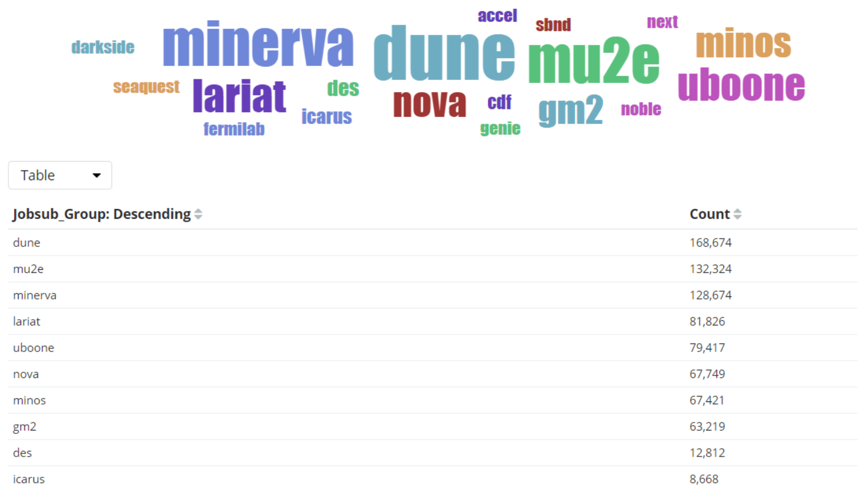
The result is displayed with both Grafana (count over time) and Kibana (total count):

**Grafana**



| | total |
|---|---|
| minerva | 198.7 K |
| dune | 187.2 K |
| gm2 | 180.8 K |
| nova | 120.3 K |
| mu2e | 117.8 K |
| minos | 101.1 K |
| uboone | 84.0 K |
| lariat | 28.8 K |
| marsmu2e | 15.9 K |
| annie | 10.0 K |
| des | 8.2 K |
| cdf | 3.1 K |
| icarus | 2.4 K |
| marsgm2 | 2.0 K |
| sbnd | 1.4 K |

**Kibana**



| Jobsub_Group: Descending | Count |
|---|---|
| dune | 168,674 |
| mu2e | 132,324 |
| minerva | 128,674 |
| lariat | 81,826 |
| uboone | 79,417 |
| nova | 67,749 |
| minos | 67,421 |
| gm2 | 63,219 |
| des | 12,812 |
| icarus | 8,668 |

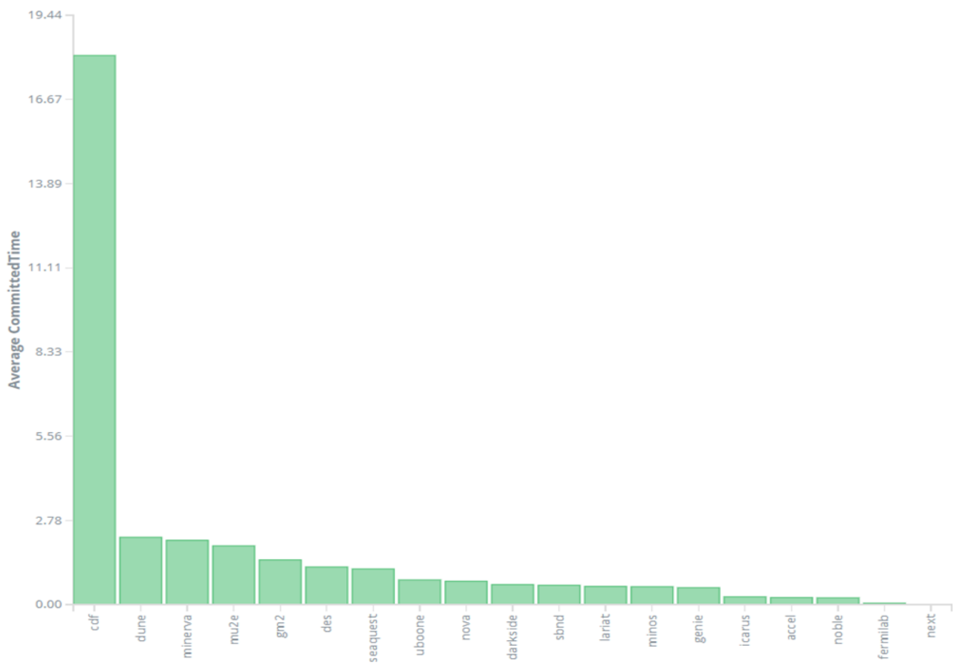
4. **What was the average walltime?**

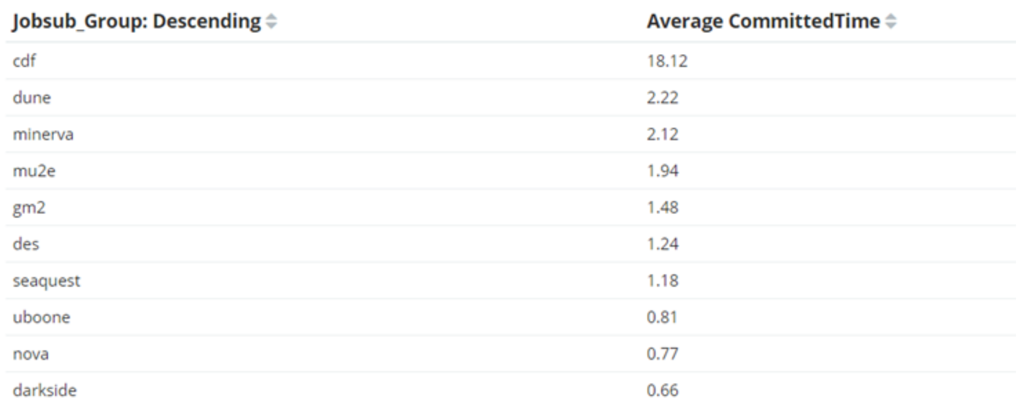
Source of the data: fifebatch-history-*

Aggregation by: Jobsub_Group

The result displayed on Kibana is the average *Committed Time (wall time)* for every experiment.

The *wall time* is the actual time taken from the start of a computer program to the end. In other words, it is the difference between the time at which a task finishes and the time at which the task started on the Grid.





| Jobsub_Group: Descending | Average CommittedTime |
|---|---|
| cdf | 18.12 |
| dune | 2.22 |
| minerva | 2.12 |
| mu2e | 1.94 |
| gm2 | 1.48 |
| des | 1.24 |
| seaquest | 1.18 |
| uboone | 0.81 |
| nova | 0.77 |
| darkside | 0.66 |

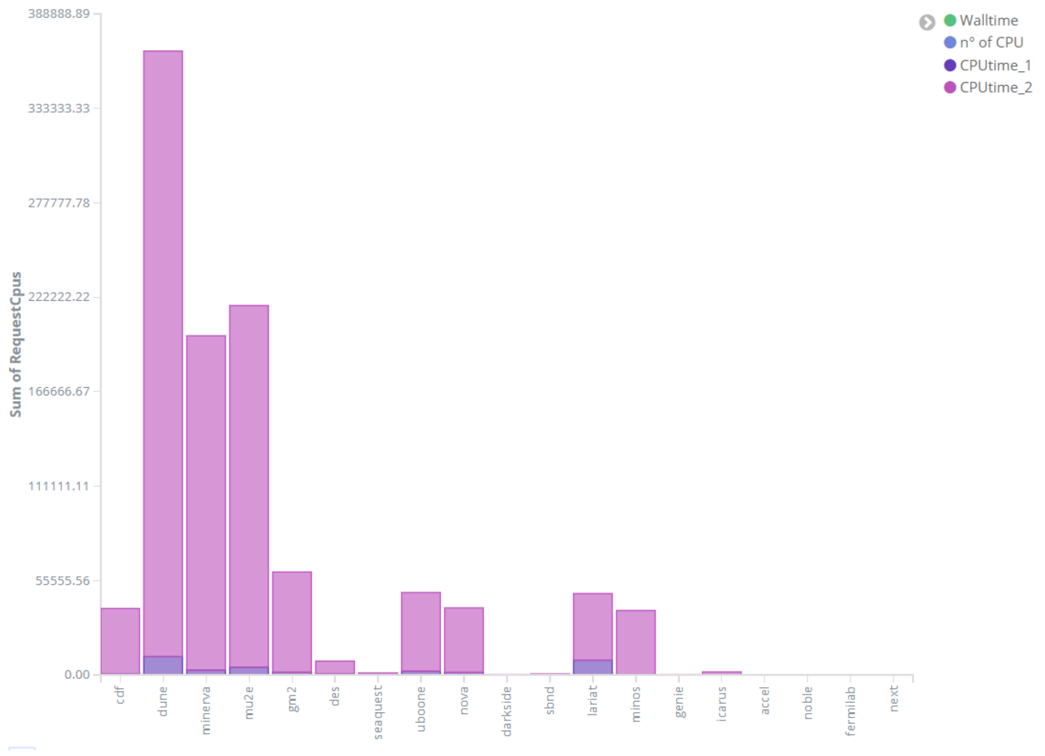
5. **What was the overall CPU efficiency for all jobs $\left(\frac{cputime * n^\circ cores}{walltime}\right)$ ?**

Source of the data: fifebatch-history-*

Aggregation by: Jobsub_Group

This result needed the combination of three different information:

- CpuTime: how much time (seconds) the CPU was busy
- N° Cores: number of cores of the Grid's nodes
- Walltime: average time (seconds) in which the experiment was running on the Grid.
-



**mu2e**

CommittedTime_dict : 256959.99

RequestCpus_dict : 1.0

RemoteSysCpu_dict : 4444.66

RemoteUserCpu_dict : 212680.43

**dune**

CommittedTime_dict : 374501.34

RequestCpus_dict : 1.42

RemoteSysCpu_dict : 10852.28

RemoteUserCpu_dict : 356118.46

…

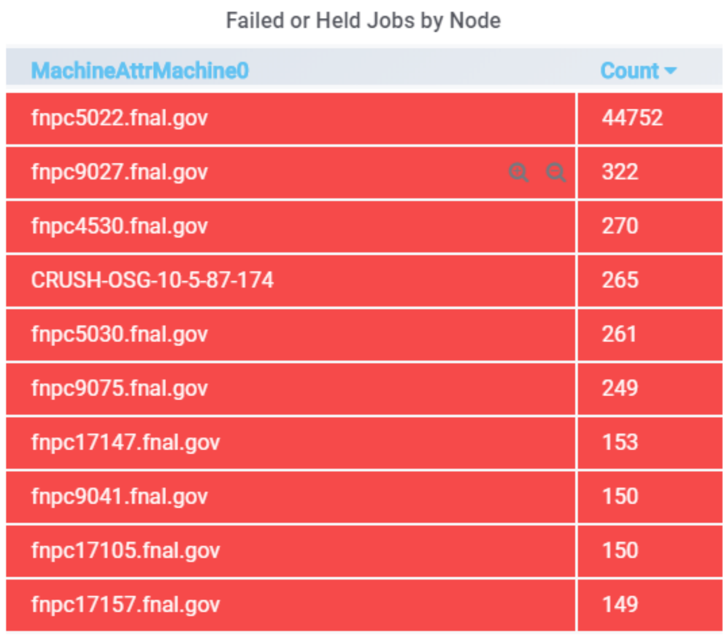
**EFFICIENCY:**

| | |
|---|---|
| dune | 1.4510446440455007 |
| minerva | 1.367907158302796 |
| mu2e | 1.1834652621299644 |
| gm2 | 1.5771239620829476 |
| uboone | 1.3327282482045406 |
| cdf | 1.485566431028936 |
| nova | 1.3144842954757536 |
| lariat | 1.016526965315123 |
| minos | 1.0473985215982602 |
| des | 2.4452951469652624 |

For this reason it was used the low-level *Elasticsearch* library to get the data from Elasticsearch's server: this allows to write complex queries and they can be used in different languages/libraries, e.g Kibana.
The result displayed is the output of the code written in Python

## 3.2 Challenge 2

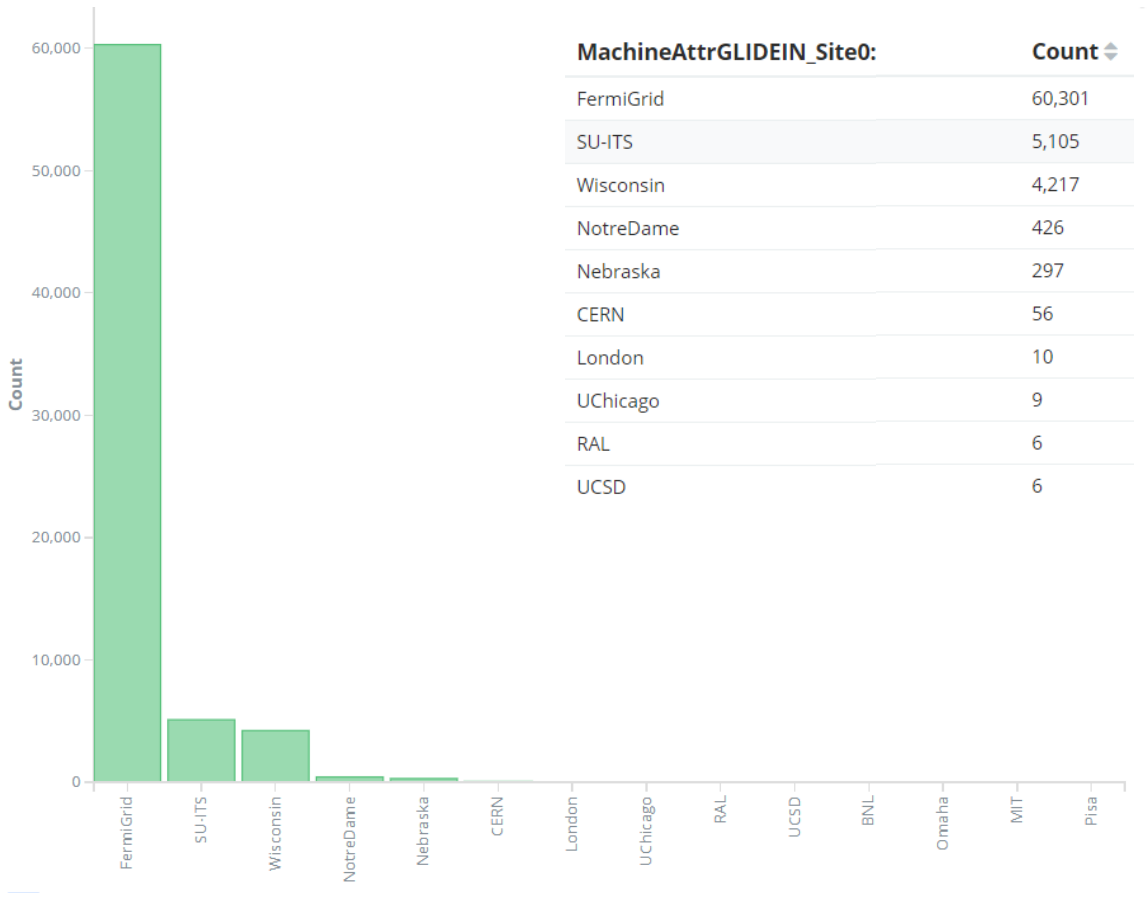
1. **What node had the most failed jobs in the past week?**
   Source of the data: fifebatch-history-*
   Aggregation by: MachineAttrMachine0


Failed or Held Jobs by Node

| MachineAttrMachine0 | Count |
|---|---|
| fnpc5022.fnal.gov | 44752 |
| fnpc9027.fnal.gov | 322 |
| fnpc4530.fnal.gov | 270 |
| CRUSH-OSG-10-5-87-174 | 265 |
| fnpc5030.fnal.gov | 261 |
| fnpc9075.fnal.gov | 249 |
| fnpc17147.fnal.gov | 153 |
| fnpc9041.fnal.gov | 150 |
| fnpc17105.fnal.gov | 150 |
| fnpc17157.fnal.gov | 149 |

## 3.3 Challenge 3

1. **What site had the most disconnected jobs in the past week?**
   Source of the data: fifebatch-events-*
   Aggregation by: MachineAttrGLIDEIN_Site0
   The jobs were filtered by "event type number 22" that indicates an event of disconnection.



| MachineAttrGLIDEIN_Site0: | Count |
|---|---|
| FermiGrid | 60,301 |
| SU-ITS | 5,105 |
| Wisconsin | 4,217 |
| NotreDame | 426 |
| Nebraska | 297 |
| CERN | 56 |
| London | 10 |
| UChicago | 9 |
| RAL | 6 |
| UCSD | 6 |

## 3.4 Challenge 4

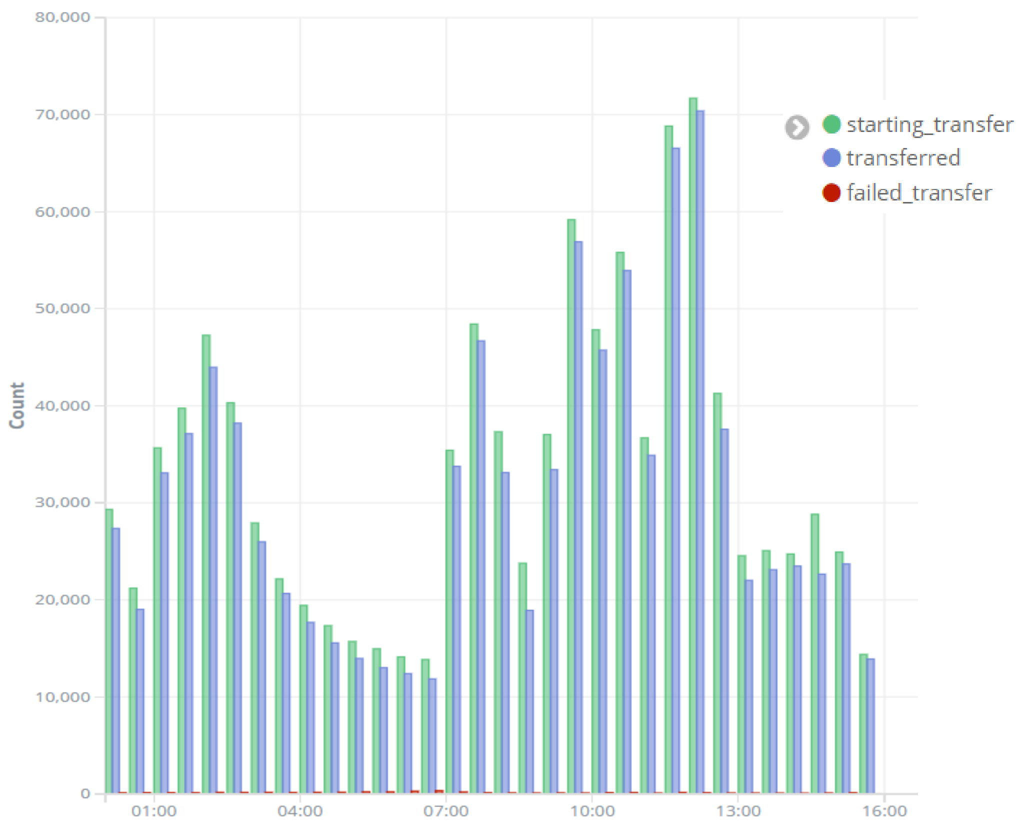
1. **How many files were transferred in the past day?**

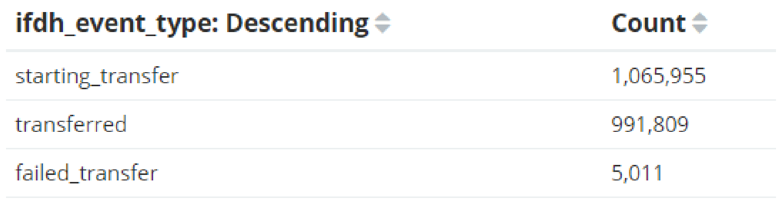
   Source of the data: fife-dh-*

   Aggregation by: ifdh_event_type

   It was considered just 3 transfer event types:

   - starting transfer
   - transferred
   - failed transfer





| ifdh_event_type: Descending | Count |
|---|---|
| starting_transfer | 1,065,955 |
| transferred | 991,809 |
| failed_transfer | 5,011 |

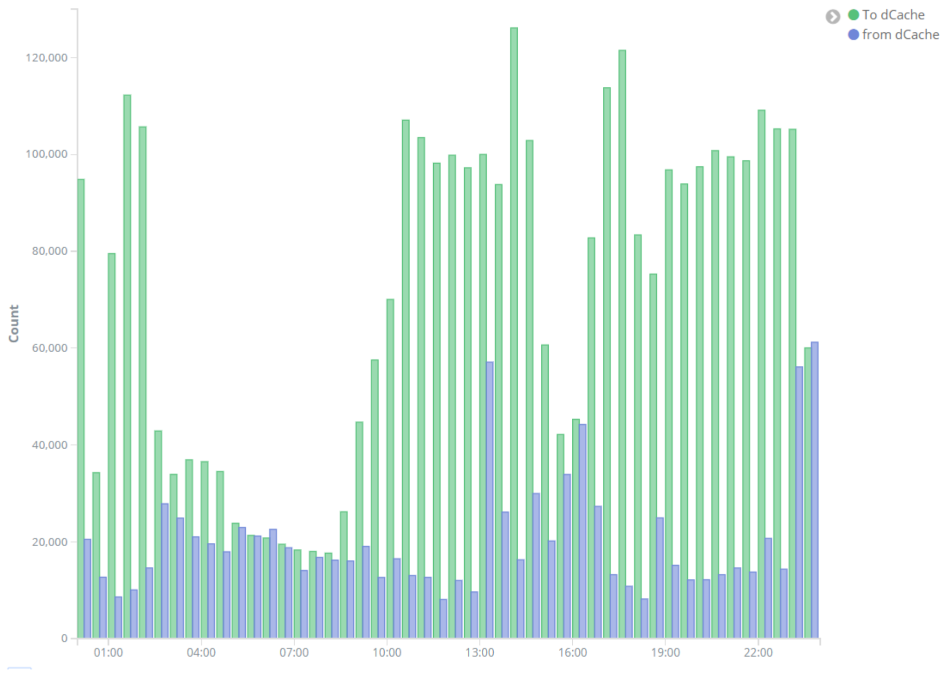
2. **How many transfers were made to/from the dCache?**

Source of the data: fife-dh-*

Data filtered by destination and source path to recognize if it’s a transfer to/from dCache (the directory of dCache is “pnfs”):

- To dCache fetered by *DestPath: *pnfs**
- From dCache filtered by *SourcePath: *pnfs**

The results are displayed on Kibana over the last day:

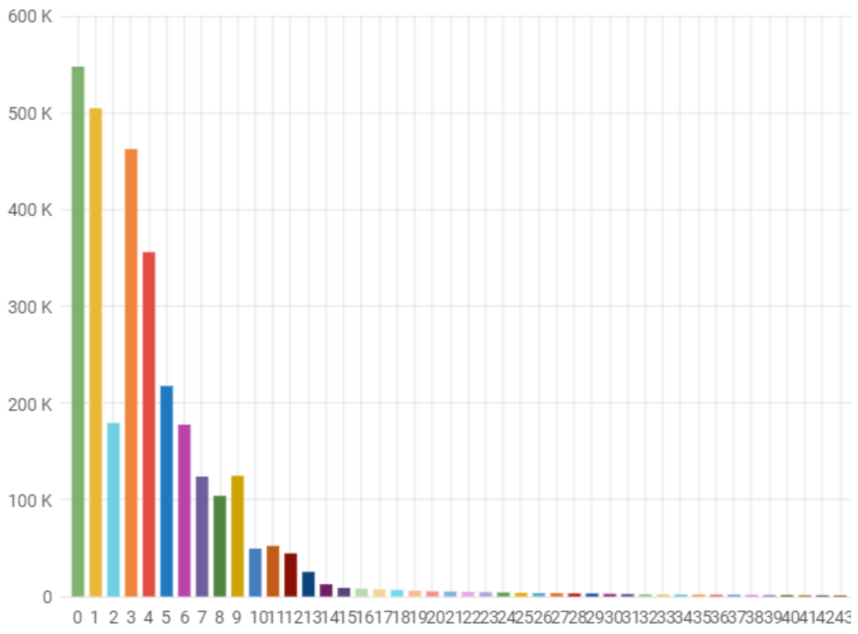


3. **Create a histogram of transfer times**

   Source of the data: fife-dh-*

   Aggregation by: transfer_time

   The graph on Grafana displays the how many jobs (y-axes) took a certain amount of time (x-axis, in seconds) to be fully transferred in the last day. We can see that most of the jobs are transferred within 14 seconds.



## 3.5 Challenge 5

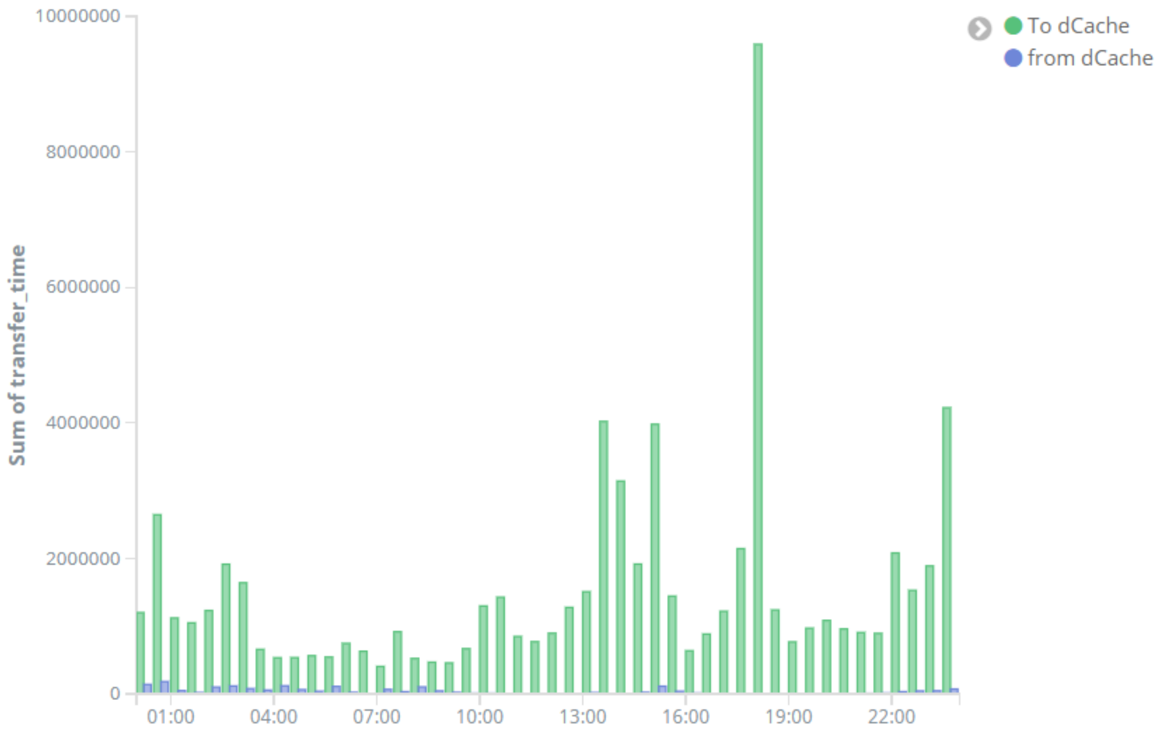
**1. For a given job, calculate how much time was spent transferring input/output to job**

Source of the data: fife-dh-*

Data filtered by destination and source path to recognize if it's a transfer to/from dCache:

- To dCache filtered by *NOT SourcePath:*pnfs**
- From dCache filtered by *SourcePath:*pnfs**

The result displayed is a graph in Kibana that shows both information:

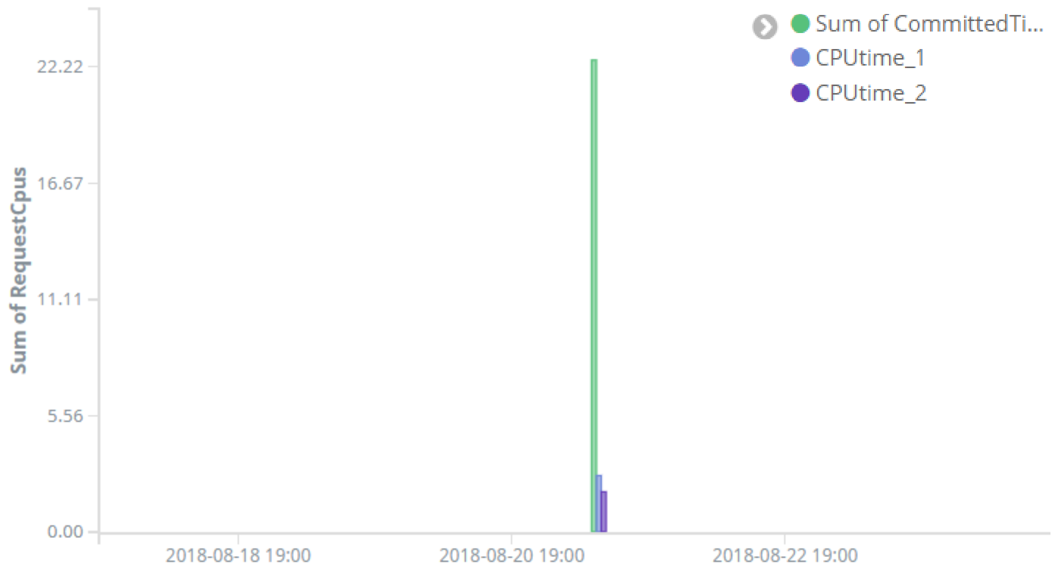


**2. For a given job, calculate how much time was spent running (CPU time) and the Walltime**

Source of the data: fifebatch-history-*

The *wall time* is the actual time taken from the start of a computer program to the end.

The CPU time is the time in which the job was actually running on a CPU. It's divided in RemoteSysCpu (CPUtime_1) and RemoteUserCpu (CPUtime_2)



3. **For a given job, calculate how much time was wasted**

   Source of the data:

   fifebatch-history-* for *CPU time* and *walltime*

   fife-dh-* for the *transfer time*

   The formula to calculate the wasted time is:

$$Wasted\_time = Walltime - (CPU\ time + Transfer\_Time)$$

   The output is an example for a specific job (JobsubJobId: 10834906):

```
CommittedTime in seconds, CpuTime in seconds, transfer_time in seconds:

CommittedTime: 20364.0
RemoteSysCpu : 35.0
RemoteUserCpu: 20198.0
transfer_time: 86.64

Wasted time in seconds:  44.36
```

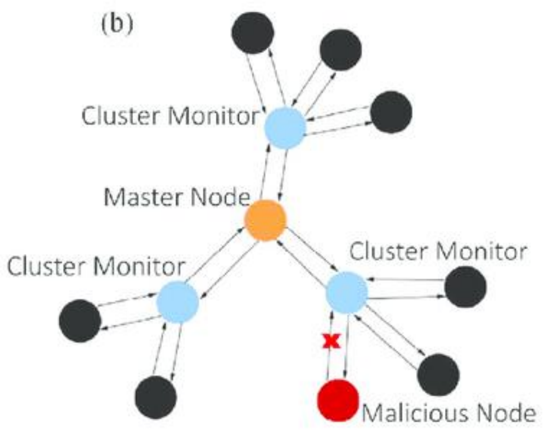
# 4. Blackhole Detection

The aim of this section is to develop a program that identifies potential black hole nodes and sends an alert.

## 4.1 What is a Blackhole

We have two possible definitions for a *blackhole:*

1. Nodes or sites with high number of failed, held or disconnected jobs that **are not user error**. We don't want to start more jobs on these nodes because they are malfunctioning.
2. Job clusters with high failure rate **due to user error**. We don't want to start more jobs on these nodes since they will just waste resources.



## 4.2 How to detect a Blackhole

Indications of a Blackhole node:

- High rate of failed jobs: Exit code ≠ 0
- High rate of held jobs: Jobs not finished for an error
- High rate of disconnections: Data exchange interrupted
- dCache issues: Transfer Fails
- No successful jobs recently: a high rate of held and failed jobs without any successful jobs

The transfer fails will not be considered in this analysis.

## 4.3 Data Exploration & Features Extraction

To find the best data suitable for the analysis it was necessary to explore every index type and field in the memory database.

The *fifebatch-events* index contains all the data necessary for a preliminary approximation: it has HTCondor event logs collected in real time.

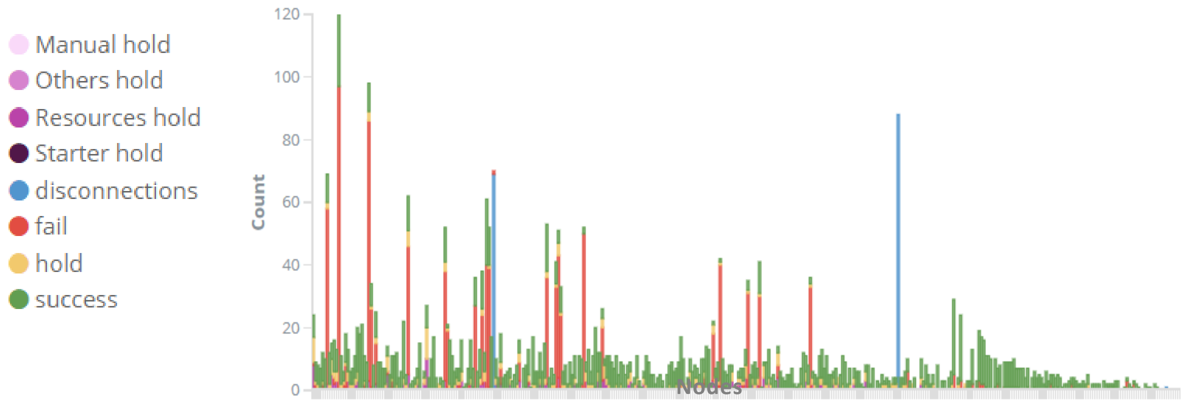
The features extracted from this index of memory are:

- How many **succeeded** jobs we had
- How many **failed** jobs we had
- How many **held** jobs we had, divided into different "hold reason" codes
- How many **disconnections** we had

The hold reasons were divided because some of them are more crucial than others:

- **Manual Hold** reason: the user manually hold his jobs, it is usually not an issue;
- **Resources Hold** reason: the jobs are hold because they are exceeding the limit of the resources assigned (memory available, too much processing time, number of CPU etc...);
- **Starter Hold** reason: the jobs do not start on the worker node, it is usually the biggest issue;
- **Other Hold** reasons: this groups all the other hold reason codes.

So, the final **features extracted** are listed in the following figure:



## 4.4 Blackhole Node Detection: approach

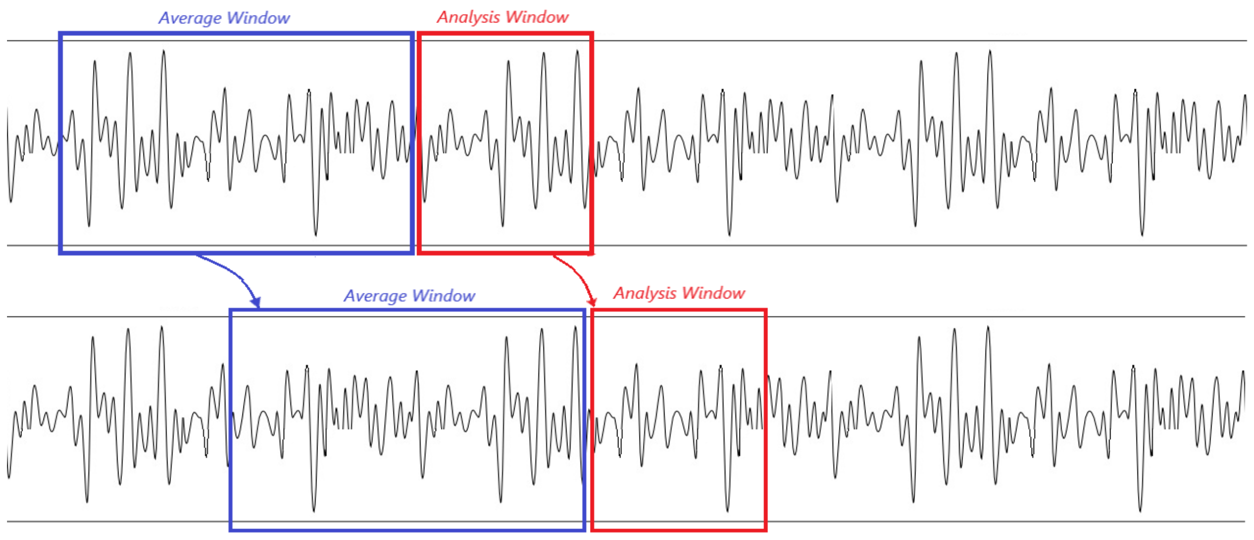
For the detection of the *blackholes* it was implemented a double “sliding window” over the entire day. We consider a time window for the analysis of the data and another time window as a reference point. Once the analysis is terminated both windows are moved forward in time.

1. **Analisys Window:**

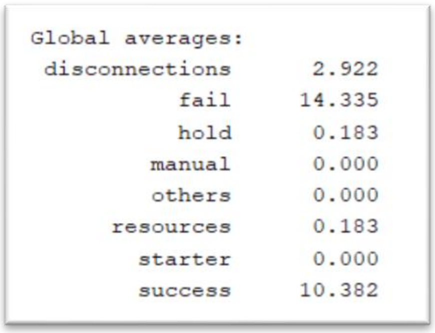
   In this short time interval the features listed before are analyzed for each node in the grid to detect aberrant conditions. Produces a ‘’total count’’ for each feature. The time interval selected for the analysis is 30 minutes.

2. **Average Window:**

   This longer time interval is used to calculate the average of each feature in order to create a reference for the analysis window. In particular, it produces a reference value called the *Global Average* for each feature: it’s the average computed considering the values from all the nodes. The time interval selected for the average is 2 hours.



An example of *Global Average* computed for a generic node is shown in the following figure:



```
Global averages:
 disconnections       2.922
           fail      14.335
           hold       0.183
         manual       0.000
         others       0.000
      resources       0.183
        starter       0.000
        success      10.382
```

## 4.5 Blackhole Node Detection: attempts

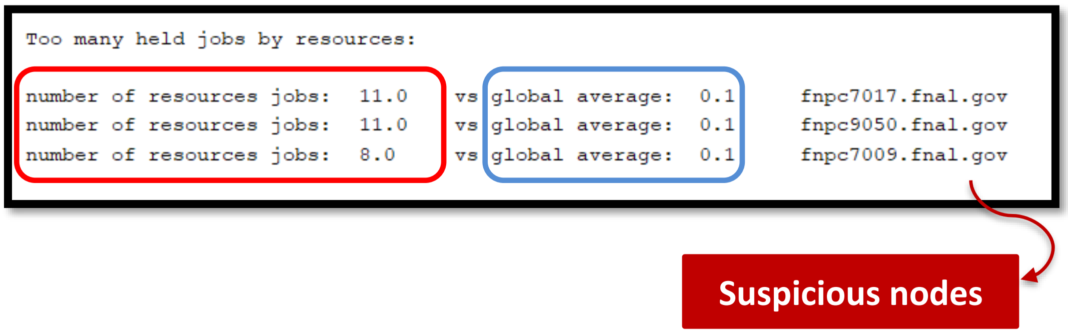
### 1. First Attempt:

As a first try was set a threshold (in this case the *global average*) for each feature.

If a node gets over this value it was printed the corresponding problem.

$$if\ total\ count(feature) > global\ average(feature):\ \boldsymbol{Trigger}!$$

Here is an example of the python script's output that performs this analysis: we can see three nodes that had 11 or 8 jobs held with a resource error versus an average (threshold) of 0.1. All these nodes were labelled as possible blackholes.



```
Too many held jobs by resources:

number of resources jobs:   11.0    vs global average:   0.1      fnpc7017.fnal.gov
number of resources jobs:   11.0    vs global average:   0.1      fnpc9050.fnal.gov
number of resources jobs:   8.0     vs global average:   0.1      fnpc7009.fnal.gov
```

The problem of this method is that it considers each feature separately, it doesn't combine them all.

## 2. Second Attempt:

This attempt tries to make an analysis that considers all the features combined.

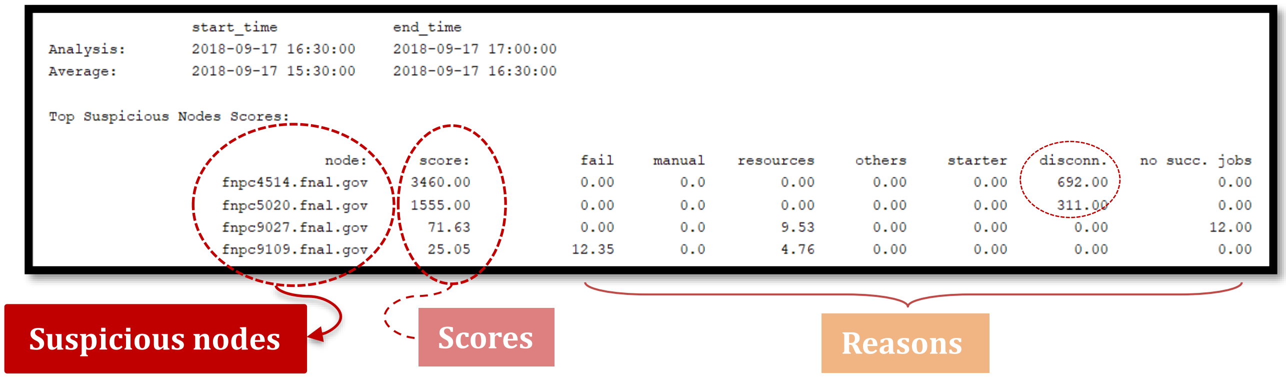
It was computed a **weighted average** for each node over all his features in this way:

- It was given to each node and feature a $\boldsymbol{score} = \frac{total\ count(feature)}{global\ average(feature)}$ ;
- It was then given each score a **weight** and it was made a linear combination of all these values.

$$Total\_Score = \sum_{features} \boldsymbol{score}(feature) * \boldsymbol{weight}(feature)$$

Here is an example of the python script's output that performs this analysis: from the left to the right there is the list of suspicious nodes, than the list of the scores and finally the reason of the scores.

We can see two nodes have a very high score (respectively 3460 and 1555) that is due to a high rate of disconnected jobs, and the node fnpc9027 that has a score of 71 because it has not any successful job. All these nodes were labelled as possible blackholes.



```
                start_time            end_time
Analysis:       2018-09-17 16:30:00   2018-09-17 17:00:00
Average:        2018-09-17 15:30:00   2018-09-17 16:30:00

Top Suspicious Nodes Scores:

                            node:   score:        fail   manual   resources   others   starter   disconn.   no succ. jobs
               fnpc4514.fnal.gov   3460.00        0.00      0.0        0.00     0.00      0.00     692.00            0.00
               fnpc5020.fnal.gov   1555.00        0.00      0.0        0.00     0.00      0.00     311.00            0.00
               fnpc9027.fnal.gov     71.63        0.00      0.0        9.53     0.00      0.00       0.00           12.00
               fnpc9109.fnal.gov     25.05       12.35      0.0        4.76     0.00      0.00       0.00            0.00
```

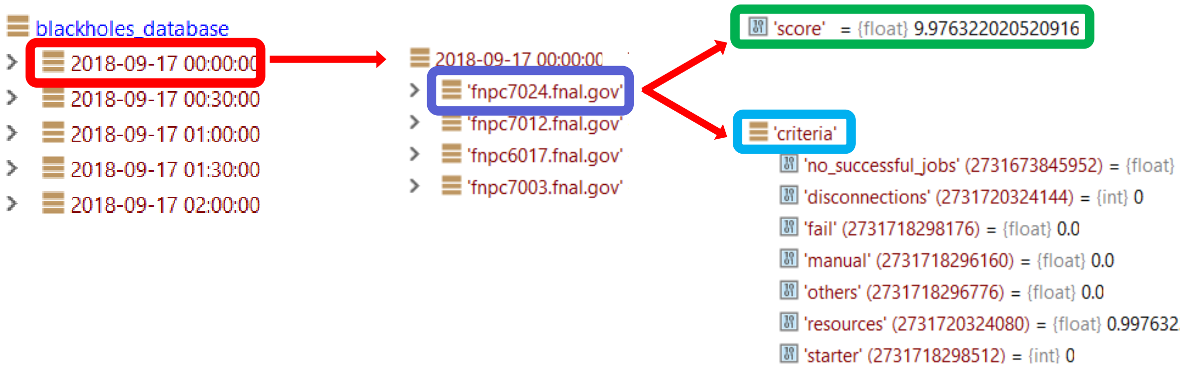
This was the analysis type implemented and then iterated over more hours/days. The code used for the analysis is shown in section number 8.

## 4.6 Blackholes Database

The full list of the suspicious nodes was saved in a unique dictionary called *blackholes_database* structured as following:

- Intervals of **time** analyzed;
- Suspicious **node list** inside a certain interval of time;
- **Score** of each node and reason (**criteria**) of the score;

This structure is shown in the figure below:



```
blackholes_database
> 2018-09-17 00:00:00
> 2018-09-17 00:30:00
> 2018-09-17 01:00:00
> 2018-09-17 01:30:00
> 2018-09-17 02:00:00

2018-09-17 00:00:00
> 'fnpc7024.fnal.gov'
> 'fnpc7012.fnal.gov'
> 'fnpc6017.fnal.gov'
> 'fnpc7003.fnal.gov'

'score' = {float} 9.976322020520916
'criteria'
    'no_successful_jobs' (2731673845952) = {float}
    'disconnections' (2731720324144) = {int} 0
    'fail' (2731718298176) = {float} 0.0
    'manual' (2731718296160) = {float} 0.0
    'others' (2731718296776) = {float} 0.0
    'resources' (2731720324080) = {float} 0.997632
    'starter' (2731718298512) = {int} 0
```

# 5. Results

In this section are reported some examples of the results obtained with this implementation.

The threshold for the score was set to 100 to minimize the number of false positives.

The time for the analysis was 30 minutes and the the time for the average was 2 hours.

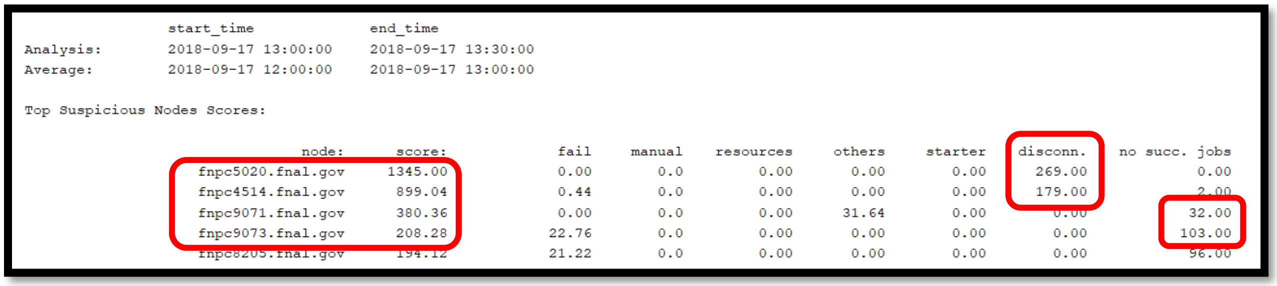
The weights used are the following:

```
scores_weight={'disconnections': 5,
               'fail': 0.1,
               'manual': 0,
               'others': 10,
               'resources': 10,
               'starter': 500,
               'no_successful_jobs': 2
               }
```

- **17th September** there were four confirmed blackholes:

  The nodes fnpc5020 and fnpc4514 had a very high score due to an high disconnection rate (269 and 179 jobs disconnected in just 30 minutes)

  The nodes fnpc9071 and fnpc9073 had zero successful jobs in the 30 minutes of analysis and respectively 32 and 103 jobs were held.



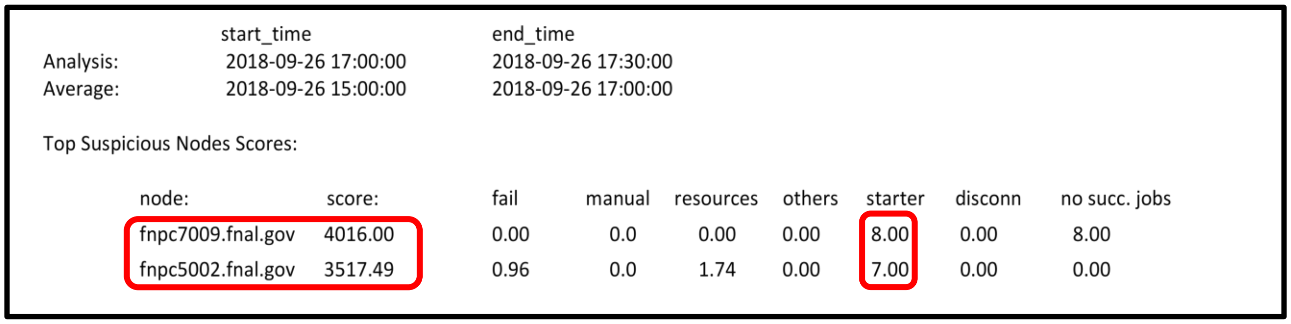
```
                  start_time             end_time
Analysis:         2018-09-17 13:00:00    2018-09-17 13:30:00
Average:          2018-09-17 12:00:00    2018-09-17 13:00:00

Top Suspicious Nodes Scores:

                            node:      score:         fail    manual   resources    others    starter   disconn.   no succ. jobs
                fnpc5020.fnal.gov     1345.00         0.00       0.0        0.00      0.00       0.00     269.00            0.00
                fnpc4514.fnal.gov      899.04         0.44       0.0        0.00      0.00       0.00     179.00            2.00
                fnpc9071.fnal.gov      380.36         0.00       0.0        0.00     31.64       0.00       0.00           32.00
                fnpc9073.fnal.gov      208.28        22.76       0.0        0.00      0.00       0.00       0.00          103.00
                fnpc8205.fnal.gov      194.12        21.22       0.0        0.00      0.00       0.00       0.00           96.00
```

- **26th September** there were four confirmed blackholes:

  In this case we had 2 blackholes due to several held jobs with a starter error, that is usually the most weighted type of error: in fact the scores of the nodes are over 3k.



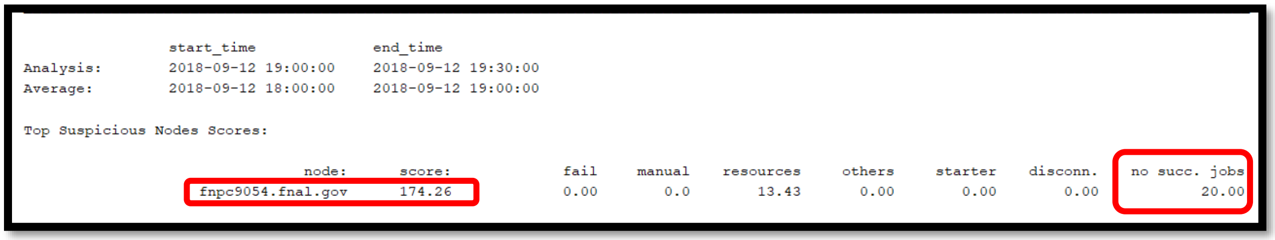
```
              start_time             end_time
Analysis:     2018-09-26 17:00:00    2018-09-26 17:30:00
Average:      2018-09-26 15:00:00    2018-09-26 17:00:00

Top Suspicious Nodes Scores:

        node:                score:     fail    manual   resources   others   starter   disconn   no succ. jobs
        fnpc7009.fnal.gov    4016.00    0.00    0.0      0.00        0.00     8.00      0.00      8.00
        fnpc5002.fnal.gov    3517.49    0.96    0.0      1.74        0.00     7.00      0.00      0.00
```

- **12th September** there was one confirmed blackhole:

  The node fnpc9054 had zero successful jobs in the 30 minutes of analysis and 20 jobs were held for the lack of resources. Even if the numbers are significantly smaller than before, the algorithm recognized this blackhole.



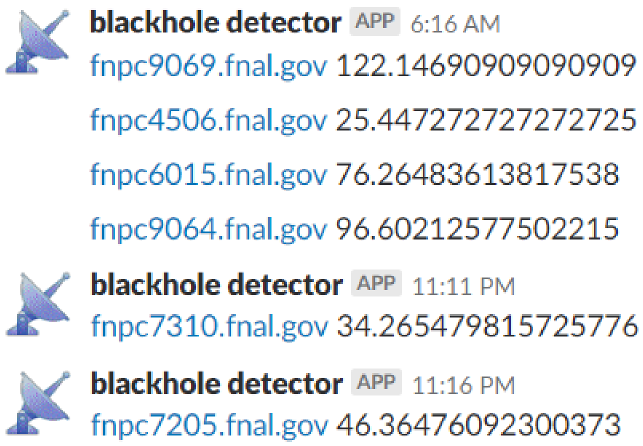
```
                 start_time             end_time
Analysis:        2018-09-12 19:00:00    2018-09-12 19:30:00
Average:         2018-09-12 18:00:00    2018-09-12 19:00:00

Top Suspicious Nodes Scores:

                              node:      score:            fail    manual    resources    others    starter    disconn.    no succ. jobs
                    fnpc9054.fnal.gov    174.26            0.00       0.0        13.43      0.00       0.00        0.00            20.00
```

## 6. Implementation

The algorithm created was then implemented to run on a *Slack* channel in real-time. The scan for any new suspicious node is repeated every 5 minutes and in the output is reported just the final score of the suspicious nodes:



**blackhole detector** APP 6:16 AM
fnpc9069.fnal.gov 122.14690909090909
fnpc4506.fnal.gov 25.447272727272725
fnpc6015.fnal.gov 76.26483613817538
fnpc9064.fnal.gov 96.60212577502215
**blackhole detector** APP 11:11 PM
fnpc7310.fnal.gov 34.265479815725776
**blackhole detector** APP 11:16 PM
fnpc7205.fnal.gov 46.36476092300373

This is just an alert and the nodes shall be checked manually to verify if they are effectively malfunctioning.

# 7. Future work

There are a lot of things in this algorithm that need to be improved:

1. **A more accurate training:**

   There are some parameters that could be adjusted to make the algorithm more precise, for example the **weights** of the weighted average, the **windows sizes** (for both analysis and reference) and the **threshold** for the detection.

2. **Integration of a control over transfer fails:**

   Until now we considered a very small set of possible blackholes because the anomalies considered are related to failed jobs, held jobs and disconnected jobs. It's possible to extend the analysis over possible **transfer fails**, that are usually dCache problems, and over the **transfer time** (that shouldn't be too long)

3. **Integration of a control over sites and users:**

   The algorithm could be trained also to reveal faulty sites and users: for the moment it is considering only suspicious nodes.

# 8. Code

```
# LIBRARIES
from elasticsearch import Elasticsearch
from elasticsearch_dsl import Search,Q
from operator import itemgetter
import datetime
from collections import OrderedDict

# GLOBAL VARIABLES
global term_to_search

# PARAMTERS:
numebr_of_nodes_analyzed = 1000

# SCORES WEIGHTS:
score_threshold = 30   # Tolerance
scores_weight={'disconnections': 5,
               'fail': 0.1,
               'manual': 0,
               'others': 10,
               'resources': 10,
               'starter': 500,
               'no_successful_jobs': 2
               }

# TIME VARIABLES:
time_analysis =30
year = 2018
month = 9
days_range=range(12,28)
hours_range=range(0,24)
minutes_range=range (0,60,time_analysis)

scaling_factor = 0.25  # Sets time for the average_window=time_analysis/scaling_factor
GMT = 5               # It's -5 hours for Chicago


################################################### Functions Definition
######################################################


def set_time_intervals(year_temp, month_temp, day_temp, hour_temp, minutes_temp):

    global start_time_analysis
    global end_time_analysis
    global end_time_average
    global start_time_average

    # TIME FOR ANALYSIS
    start_time_analysis = datetime.datetime(year_temp, month_temp, day_temp,
hour_temp,minutes_temp) + datetime.timedelta(hours=GMT)
    end_time_analysis = start_time_analysis + datetime.timedelta(minutes=time_analysis)

    # TIME FOR AVERAGE
    end_time_average = start_time_analysis
    start_time_average = end_time_average - datetime.timedelta(minutes=time_analysis /
scaling_factor)

    # return start_time_analysis, end_time_analysis, start_time_average, end_time_average

def get_elasticsearch_data(term_to_search, query):
    """
    This function creates the "global average" and two dictionaries:
    - The "analysis_dictionary" collects the data for the analysis
    - The "average_dictionary" collects the data in order to create a reference point
    the average_dictionary needs to be normalized because it takes in account a time interval
much bigger compared to the analysis time (look at the "scale" variable)

    :param term_to_search   : The data will be aggregated according to this term. Type
"MachineAttrMachine0" to aggregate by nodes, type "MachineAttrGLIDEIN_Site0" to aggregate by
Sites
    :param query            : you can pass here a specific query, for example with
"MachineAttrMachine0: fnpc17146.fnal.gov" you will get only informations about that specific node

    :var analysis_dictionary : is the dictionary of nodes under analysis. it takes the time
interval between (start_time_analysis ; end_time_analysis)
    :var average_dictionary  : is the dictionary of average features of each node. it takes the
time interval between (start_time_average ; end_time_average)
    :var global_average      : is a dictionary in which we save the average value for every
feature. every average value considers the data from all the nodes

    :return             : the function returns these three dictionaries:
analysis_dictionary,average_dictionary, global_average

    """

    # 1: analysis_dictionary is the dictionary of nodes under analysis
    analysis_dictionary = get_fifebatch_events(start_time_analysis, end_time_analysis,
'fifebatch-events-*', term_to_search, query=None)

    # 2: average_dictionary is the dictionary of average features of each node in a previous time
interval
    average_dictionary = get_fifebatch_events(start_time_average, end_time_average, 'fifebatch-
events-*', term_to_search, query=None)
    # Normalization: multiplies by scaling factor (to compensate the fact that the time window is
bigger = much more informations)
    for node, stats in average_dictionary.items():
        stats.update((k, v * scaling_factor) for k, v in
                     average_dictionary[node].items())  # Scaling factor is for normalization
over time
    # Compute global average of nodes values
    global_average = calculate_global_average(average_dictionary)

    return analysis_dictionary, average_dictionary, global_average

def get_fifebatch_events(start, end, fife_index, term_to_search, query=None):  ##initialization
between brackets
    '''
    This function makes a query to elasticsearch server

    :param start           : start time for the query to elasticsearch
    :param end             : end time for the query to elasticsearch
    :param fife_index      : string that provides the index name in which i'm searching the data
    :param term_to_search : The data will be aggregated according to this term. Type
"MachineAttrMachine0" to aggregate by nodes, type "MachineAttrGLIDEIN_Site0" to aggregate by
Sites
    :param query           : you can pass here a specific query, for example with
"MachineAttrMachine0: fnpc17146.fnal.gov" you will get only informations about that specific node

    :var number_of_nodes_analyzed : it's the number of nodes that will be analyzed
    :var query_events : JSON query to the elasticsearch server

    :return complete : returns a dictionary with the full list of nodes and their features
    '''
    query_events = {
        "size": 0,
        "query": {
            "bool": {
                "must": [
                    {
                        "match_all": {}
                    },
                ],
                'filter': [{'range': {'@timestamp': {"gte": start, "lte": end}}}],
                "must_not": []
            }
        },
        "_source": {
            "excludes": []
        },
        "aggs": {
            "node": {
                "terms": {
                    "field": term_to_search,
                    "size": numebr_of_nodes_analyzed,
                    "order": {
                        "_count": "desc"
                    }
                },
                "aggs": {
                    "status": {
                        "filters": {
                            "filters": {

                                "hold": {
                                    "query_string": {
                                        "query": "MyType:JobHeldEvent",
                                        "analyze_wildcard": True
                                    }
                                },
                                "fail": {
                                    "query_string": {
                                        "query": "MyType:JobTerminatedEvent AND NOT
ReturnValue:0",
                                        "analyze_wildcard": True
                                    }
                                },
                                "success": {
                                    "query_string": {
                                        "query": "MyType:JobTerminatedEvent AND ReturnValue:0",
                                        "analyze_wildcard": True
                                    }
                                },
                                "disconnections": {
                                    "query_string": {
                                        "query": "MyType: JobReconnectFailedEvent",
                                        "analyze_wildcard": True
                                    }
                                },
                                "manual": {
                                    "query_string": {
                                        "query": "HoldReasonCode: 1",
                                        "analyze_wildcard": True
                                    }
                                },
                                "resources": {
                                    "query_string": {
                                        "query": "HoldReasonCode: (34 26)",
                                        "analyze_wildcard": True
                                    }
                                },
                                "starter": {
                                    "query_string": {
                                        "query": "HoldReasonCode: 6",
                                        "analyze_wildcard": True
                                    }
                                },
                                "others": {
                                    "query_string": {
                                        "query": "MyType:JobHeldEvent AND (NOT HoldReasonCode: (1
6 26 34) )",
                                        "analyze_wildcard": True
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }

    if query is not None:
        query_events['query']['bool']['must'] = {'query_string': {'query': query}}
    r = client.search(fife_index, body=query_events)
    #     print (r , '\n\n')
    complete = {}
    for node in r['aggregations']['node']['buckets']:
        # print vo
        complete[node['key']] = {}
        for status, stats in node['status']['buckets'].items():
            complete[node['key']][status] = stats['doc_count']
    # print(complete)
    return complete

def calculate_global_average(dictionary):
    '''
    This function computes the global average.
    We get a dictionary from the input and then:
    - we sum up the values of every node for every feature
    - we divide the sum obtained by the number of the nodes (:var lenght)

    :param dictionary : it's the dictionary over which we calculate the global average
    :return           : we obtain a dictionary that stores the average of every feature computed
considering the values from all the nodes. This variable obtained is useful as a pint of
reference for the analysis
    '''
    lenght = len(dictionary)
    complete = {}
    for node, status in dictionary.items():
        for bucket, value in status.items():
            complete.setdefault(bucket, 0)  # Adds the new key with default value=0 (if not
present)
            complete[bucket] += value

    for bucket, stats in complete.items():
        complete[bucket] = stats/lenght

    return complete

def create_suspicious_nodes_dictionary(analysis, threshold=0):  # We calculate the average for
every single node. use thethreshold just to set a threshold
    '''
    This function takes the dictionary "analysis" of the analysis interval of time and creates a
new dictionary that includes some new entries:
    "score" and "no_successful_jobs". it also moves all the features inside the "criteria" entry.
    We still need to compute the scores, so it is still 0.

    :param analysis: it's the dictionary on which we want to work
    :return: a dictionary with all the nodes, scores and criteria for the scores. We still need
to compute the scores, so it is still 0
    '''
    result ={}
    for node, stats_analysis in analysis.items():
        for aggr, value_analysis in stats_analysis.items():
            # if (value_analysis > threshold[aggr]):    # THRESHOLD
            if node not in result:
                result[node] = {}
                result[node]['criteria'] = {}
                result[node]['criteria']['no_successful_jobs'] = float(0)
                result[node]['score'] = float(0)
            if aggr != 'success' and aggr != 'hold':    ############################### EXCLUDES
```

```
SUCCESS and HOLD aggregation
                result[node]['criteria'][aggr] = value_analysis

        if analysis[node]['success']==0:
            result[node]['criteria']['no_successful_jobs'] =
float(analysis[node]['hold']+analysis[node]['fail'])

    return result

def calculate_score_suspicious_nodes(suspicious_dict,average):
    '''
    This function computes the score for each node.
    The score is a weighted average and it takes in consideration the average value of the
features in a precedent interval of time (global_average)

    score = (count(feature)/global average(feature)) * weight
    NOTE: we don't take in account global_average when we compute the score for "starter" hold
reason and for the number of "no_successful_jobs"

    :param suspicious_dict: this is the initialized dictionary of the suspicious nodes, in wich
the scores are still 0
    :param average: it's the global_average, it's a reference point of a precedent interval of
time (longer than the analysis time, look at the "scale" variable)
    :return: it returns the same dictionary in wich the scores and the reasons of the scores
(criteria:features) are updated (features are divided by the global average, the score is
computed as a weighted average)
    '''
    for node, stats_analysis in suspicious_dict.items():
        for aggr, value_analysis in stats_analysis['criteria'].items():

            if aggr=='disconnections':
                # suspicious_dict[node]['criteria'][aggr]=value_analysis/(average[aggr]+1)
############################################################## I INSERTED AN 1 because the
average is often Zero
                suspicious_dict[node]['score'] += scores_weight[aggr] *
(suspicious_dict[node]['criteria'][aggr]) # Just INCREMENT the score

            if aggr=='fail':
                suspicious_dict[node]['criteria'][aggr] = value_analysis / (average[aggr] + 1)
                suspicious_dict[node]['score'] += scores_weight[aggr] *
suspicious_dict[node]['criteria'][aggr]  # Just INCREMENT the score

            if aggr == 'manual':
                suspicious_dict[node]['criteria'][aggr] = value_analysis / (average[aggr] + 1)
                suspicious_dict[node]['score'] += scores_weight[aggr] *
suspicious_dict[node]['criteria'][aggr]  # Just INCREMENT the score

            if aggr == 'others':
                suspicious_dict[node]['criteria'][aggr] = value_analysis / (average[aggr] + 1)
                suspicious_dict[node]['score'] += scores_weight[aggr] *
suspicious_dict[node]['criteria'][aggr]  # Just INCREMENT the score

            if aggr == 'resources':
                suspicious_dict[node]['criteria'][aggr] = value_analysis / (average[aggr] + 1)
                suspicious_dict[node]['score'] += scores_weight[aggr] *
suspicious_dict[node]['criteria'][aggr]  # Just INCREMENT the score

            if aggr == 'starter':
                # suspicious_dict[node]['criteria'][aggr] = value_analysis / (average[aggr] + 1)
                suspicious_dict[node]['score'] += scores_weight[aggr] *
suspicious_dict[node]['criteria'][aggr]  # Just INCREMENT the score

            if aggr == 'no_successful_jobs':
                suspicious_dict[node]['score'] += scores_weight[aggr] *
suspicious_dict[node]['criteria'][aggr]  # Just INCREMENT the score

    return suspicious_dict

def print_suspicious(suspicious_dict):
    '''
    This function prints the list of suspicious nodes sorted by the score

    :param suspicious_dict: dicionary to print of the suspicious nodes
    :param score_threshold: threshold for the score. default value is 1.0
    '''
    # Creating a list (node,score) just to sort by scores
    suspicious_list = list()
    for node, stats_analysis in suspicious_dict.items():
        if suspicious_dict[node]['score'] > 0:      # If we have a score > 0 we put the node on a
separate list
            suspicious_list.append([node, suspicious_dict[node]['score']])

    # Sorting the list by scores
    suspicious_list = sorted(suspicious_list,key=itemgetter(1), reverse=True)  # sorting elements
of the list

    #Printing time intervals of Analisys and Average

print('_____________________________________________________________________________________________
__________________________________________________')
    print('\n\t\t\t\t start_time \t\t\t end_time')

    print('Analysis:\t\t', start_time_analysis-datetime.timedelta(hours=GMT),'\t',
end_time_analysis-datetime.timedelta(hours=GMT),
          '\nAverage:\t\t', start_time_average-datetime.timedelta(hours=GMT),
'\t',end_time_average-datetime.timedelta(hours=GMT))

    # Printing the elements of the dict sorted by scores
    print('\nTop Suspicious Nodes Scores:\n')
    print("%37s\t%10s\t%15s\t%10s\t %10s\t%10s\t%10s\t%10s\t%15s " % ("node:", "score:", "fail",
"manual", "resources", "others","starter", "disconn.", "no succ. jobs"))
    for elem in suspicious_list:
        if elem[1]>score_threshold:
            print("%37s\t%10.2f\t%15.2f\t%10.1f\t %10.2f\t%10.2f\t%10.2f\t%10.2f\t%15.2f " %
                  (elem[0],
                   elem[1],
                   suspicious_dict[elem[0]]['criteria']['fail'],
                   suspicious_dict[elem[0]]['criteria']['manual'],
                   suspicious_dict[elem[0]]['criteria']['resources'],
                   suspicious_dict[elem[0]]['criteria']['others'],
                   suspicious_dict[elem[0]]['criteria']['starter'],
                   suspicious_dict[elem[0]]['criteria']['disconnections'],
                   suspicious_dict[elem[0]]['criteria']['no_successful_jobs'])
                  )

def get_blackholes(analysis_dictionary, global_average):
    '''
    This function sets the interval of time for the analysis and for the average
    Then it gets data from elasticsearch and creates a dictionary with the suspicious nodes
    It returns a dictionary of the suspicious nodes with their scores

    NOTE: Here we set the average_time interval through the "scale" variable
(=analysis_time/scale)
    example: if we set a scale of 0.25, the average time will be four times longer than the
analysis time

    :param end              : end time for the analysis
    :param term_to_search   : The data will be aggregated according to this term. Type
"MachineAttrMachine0" to aggregate by nodes, type "MachineAttrGLIDEIN_Site0" to aggregate by
Sites
    :param query            : you can pass here a specific query, for example with
"MachineAttrMachine0: fnpc17146.fnal.gov" you will get only informations about that specific node
    :return                 : dictionary of the suspicious nodes with their scores
    '''
    # Get suspicious nodes
    suspicious_nodes_dictionary = create_suspicious_nodes_dictionary(analysis_dictionary)

    # Calculate Scores and Sort the List
    suspicious_nodes_dictionary =
calculate_score_suspicious_nodes(suspicious_nodes_dictionary,global_average)  # it returns a list
with (node,score) and add the scores to the suspicious_nodes_dictionary

    return suspicious_nodes_dictionary

def create_blackholes_database(term_to_search, query, year, month, days_range, hours_range,
minutes_range):
    '''
    creates an ordered Dict of the suspicious nodes.
    The difference between the previous one is that now we keep track of the time at which we
detected the blackholes

    :param term_to_search  : The data will be aggregated according to this term. Type
"MachineAttrMachine0" to aggregate by nodes, type "MachineAttrGLIDEIN_Site0" to aggregate by
Sites
    :param query           : you can pass here a specific query, for example with
"MachineAttrMachine0: fnpc17146.fnal.gov" you will get only informations about that specific node
    :param year            : insert the year for the analysis time
    :param month           : insert the month for the analysis time
    :param days_range      : insert the day range for the analysis time
    :param hours_range     : insert the hours range for the analysis time
    :param minutes_range   : insert the minutes range for the analysis time
    :return                : an ordered dictionary with time division, list of suspicious nodes
and their scores
    '''

    blackholes_database = OrderedDict()
    for day in days_range:
        for hour in hours_range:
            for minutes in minutes_range:
                set_time_intervals(year, month, day, hour, minutes)
                analysis_dictionary, average_dictionary, global_average =
get_elasticsearch_data(term_to_search, query)

                blackholes = get_blackholes(analysis_dictionary, global_average)
                # print_suspicious(blackholes)

                blackholes_database.update({start_time_analysis - datetime.timedelta(hours=GMT):
blackholes})
    return blackholes_database

def print_blackholes_database(blackholes_database):
    '''
    this function prints all the suspicious nodes divided by the time ranges of analysis
    :param blackholes_database: ordered dictionary to print

    '''

    for time, blackholes_temp in blackholes_database.items():
        for node, status in blackholes_temp.items():
            if status['score'] > score_threshold: #if i have at least 1 node with score>threshold
then i do:
                set_time_intervals(time.year, time.month, time.day, time.hour, time.minute)
                print_suspicious(blackholes_temp)
                break

################################################### MANUAL RANGE
######################################################

# client = Elasticsearch('https://fifemon-es.fnal.gov', timeout=120)
#
# # TIME VARIABLES:
# year    =   2018
# month   =   9
# day     =   18
# hour    =   12
# minutes =   0
# set_time_intervals(year,month,day,hour,minutes)
# term_to_search="MachineAttrMachine0"
# analysis_dictionary, average_dictionary, global_average =
get_elasticsearch_data(term_to_search, query=None)#
#
# print('\nGlobal averages: ')
# for key, value in global_average.items():
#     print('%15s %10.3f' % (key, value))
#
# print('\nScore Threshold:\t',score_threshold)
#
# print('\nAnalysis_time:\t\t', time_analysis, 'm')   # Printing the time intervals considerated
#
#
# blackholes = get_blackholes(analysis_dictionary, global_average)
# print_suspicious(blackholes)

########################################### Blackhole Nodes intentification
############################################

client = Elasticsearch('https://fifemon-es.fnal.gov', timeout=120)

print('\nScore Threshold:\t',score_threshold)       # Printing the threshold for the score
print('\nAnalysis_time:\t\t', time_analysis, 'm')   # Printing the time intervals considerated


print('\n\n ############################################################# BLACKHOLE NODES
##############################################################')
term_to_search="MachineAttrMachine0"
query=None
score_threshold = 100   # Tolerance
blackholes_database = create_blackholes_database(term_to_search, query, year, month, days_range,
hours_range, minutes_range)
print_blackholes_database(blackholes_database)
print()
```