**Design Approach for Rule Engine:**

Below document describes about the approach and design of the Rule Engine Solution

Steps Followed:

1. Designed a config file to read the rule

Format of the rules are as given Below

#Rule Declaration File

#Format should be as below

# Rules to be followed as below

#Possible values for datatypes are integer, string, datetime, and the operators to be used for the same are as below

# >, <, >=, <=, !=, = operators can be used for integer and datetime

# =, != to be used only for String datatype

#column should have value, as inout. If the input is extended to have any other json key, the same column can be used to represent another key

#signal,column,operator,value,datatype

ATL2,value,!=,2017-08-20 07:25:29,datetime

ATL2,value,=,HIGH,string

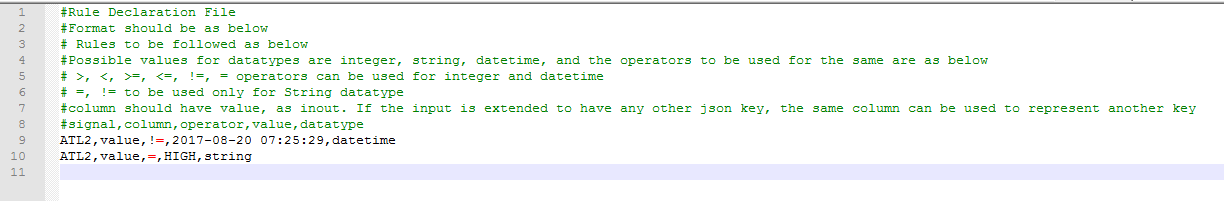
1. Reading only the lines which is not starting with a “#”. These lines are appropriate rule
2. Once the rules are read, from the file, rules.ini, these parameters will be sent to a function called createRule. This function will prepare rule in the below fashion, and returns the rule value. However, there are few error handling cases, where createRule can return -1, -2 or -3.

-1 for invalid operator, -2 for invalid datatype, -3 for conditional failure of operator in case of datatype string

* 1. ATL2,value!=2017-08-20 07:25:29#datetime
  2. The above representation is assumption, and goes as an input to the program executor

1. These rules, and the json data which is read from the input file, raw\_data.json, will be sent to another function called executeRule.
   1. Execute rule, will split the complete rule, which it received,
      1. E.g: ATL2,value,!=,2017-08-20 07:25:29,datetime
   2. After appropriate split is done on the rule to be executed, the rule will be checked for which data type to be considered and which operator to be used.
   3. For string datatype, only = and != are allowed
   4. For integer and datetime datatypes, >, <, >=, <=, =, != are applicable
   5. Result set will be formed for the inverse case. For example if the rule is value=HIGH, then all the data, which fails this criterion will be returned.
2. Example input and output as below

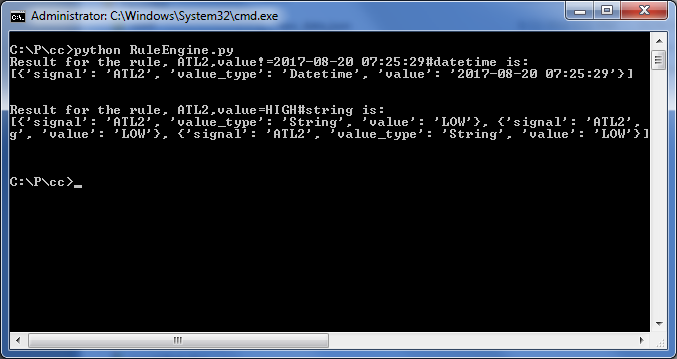
Input:



Output:

Note: rules.ini, raw\_data.json and RuleEngine.py should all be in the same directory for execution. Please use python3 for execution

Command to execute: python RuleEngine.py



**Performance of the program:**

The program, is highly performant, as much of filtering will happen on the json, even before the rules get executed. Here are the few instances, of two rules getting executed in succession



**Complexity of the program:**

The time complexity of the program is O(n), as the data has to be traversed once.

**Future Enhancements:**

The future plan of action, if more time is spent on this activity would be

1. Improve on the rule processing, something like, a rule like, value in between 20 and 30. Currently such rule is not there
2. This solution can be as generic as possible, Instead of asking the user to provide the data type for comparison, programatically, we can determine what kind of data is passed
3. Introducing a class to make the methods accessible, from other programs and to make it more object oriented, to create a perfect solution for a rule engine.
4. Once the above criterion is satisfied, the solution can be made generic, to process as input file, of any json. This would become an ideal rule engine solution
5. Also, an Apache kafka bus can be used, and the json data produces can be configured, and can have a several consumers, where each consumer can use a different rule, or a set of rules to get the data in real time, for real time analytics.