**Orleans Performance Counter Analyzer tool**

Orleans Performance Counter analyzer is a tool, which can be used to visualize, analyze and detect anomalies in performance counter data of Orleans logs. Because there are many performance counters available identifying which counters to do the analysis is challenging. The tool supports following two kinds of analysis depending on the way it choose the set of counters for further analysis.

1. **User driven analysis**

In this case user knows what are the counters need to be analyzed. Therefore the user will provide a set of counters for the analysis.

1. **Data driven analysis**

In this case user does not provide a set of counters, the tool will find set of counters which it thinks anomalous based on certain criteria. Currently tool provides two data driven analysis methods to find the send of counters for further analysis. In data driven analysis the tool will find set of counter which it thinks anomalous. Then user can define set of actions, which will describe below to do the further analysis on these counters.

1. Data Analysis

In this techniques the tool check for anomalous counters for a performance counter dataset based on the following equation. It also outputs the silo name in which the anomaly occurs. The user has to provide the threshold value.

The anomaly detection is based on the following equation.

*X = |GlobalMed - SiloMed| / (GlobalStandardDev)*

X is compared against a user provided threshold, and user can experiment with the threshold value, before deciding on the desired value.

1. Comparative analysis

In this technique the tool check for anomalous counters by comparing a particular dataset with a reference dataset. The tool outputs the anomalous counters based on the following equation. User has to provide a threshold value in this case also.

The comparative analysis between datasets is done using the following equation.

X = |ReferenceDatasetMedian – DatasetMedian|/ReferenceDatasetRange

This comparison can be global to global, silo to silo or time to time

**Actions for further analysis**

For both of the cases user has to provide a set of actions. The tool currently supports following actions.

1. **Excel based visualization**

For each counter which requires further analysis the tool provides three kinds of visualizations. A detailed surface chart shows all the values of the counter in each silo and at each time point. The Silo based chart shows, the summary statistic at each time point. The silo based chart shows the summary statistic of each

1. **Threshold analysis**

The user provides an xml file which contains set of rules. The structure of this file is defined in section “**Threshold based analysis configuration elements**” at the end of this document. Threshold Analysis can only be done with user driven analysis

1. **Correlation analysis (experimental)**

Tool can calculate Pearson, Spearman and Histogram based correlation analysis techniques. In the user driven analysis process the user will provide the kpi counters. These KPIs can be based on his domain knowledge, or they can be counters from previous threshold based analysis process. In the data driven analysis process they will be the output counters from an anomaly detection. For Pearson and Spearman coefficients the tool outputs the explanatory counters for any given KPI, if the corresponding correlation value is greater than 0.9. Tool can also provide histogram explanation. This is based on the work done by Christian Konig at XCG.

1. **Variance analysis (experimental)**

Tool can calculate and check whether the variance of the normalized data is above a certain threshold and output those counters.

1. **Performance tuning (experimental)**

Tool can aggregate counter values based on stages the user provides. This feature is still in an experimental one, so the user must provide stage definitions in PerformanceTuningAnalyzer.cs. The definitions should be moved into a configuration xml in future versions.

We have described all these options and how to configure it below, in the configuration section.

**Data collection methods**

Tool currently provide two methods for collecting performance counter data.

1. Log based data collector

The tool parses the performance counter values from log files of the system execution and store them in memory for further analysis.

1. Azure storage based data collector

The tool connect to the Azure storage and query the performance counter data for a particular load test.

**Running the tool**

To start the tool user has to provide a configuration file, as follows.

*OrleansStatAnalyzer.exe config\_file\_name*

If file is not provided the tool will search for the following default file.

*config\AnalysisConfiguration.xml*

Because anomaly detection is a trial and error process, the tool provided the facility for the user to provide thresholds during the anomaly detection and comparative analysis stages. User can changed the threshold values to find out the optimal threshold value based on the performance counters output by the tool. To exit this interactive process user just need to type “quit”.

**Configuration**

The functionality of the tool depends on the configuration options. There are two types of child elements included in the main configuration element “*AnalysisConfiguration*”. They are “Dataset” and “ComparativeAnalysis”.

“Dataset” represents the performance counter data of a particular experiment. This element also encloses the type of actions will be taken for the analysis of data. There can be any number of datasets present in the configuration file.

“ComparativeAnalysis” encloses the actions need to be taken to compare the datasets specified in the configuration file.

Following is description about these two elements and their child elements.

**Dataset element**

This element encapsulates a dataset and related analysis options. For each different experiment/load test user can specify different Dataset element. Following is a description of its child elements and attributes.

**Attributes**

|  |  |  |
| --- | --- | --- |
| **Attribute Name** | **Values** | **Description** |
| Reference | True (Default) / False | Whether this is a reference Dataset or not, for comparative analysis |

**Child Elements of Dataset element**

**1. StorageType**

Specifies the type of the storage of the performance counter data.

Attributes

|  |  |  |
| --- | --- | --- |
| **Attribute Name** | **Values** | **Description** |
| Name | Log | Tool will use log files to extract performance counter data. The tool will process the “Log” child element in this case |
| Azure | Tool will use Azure storage to extract the performance counter data. The tool will only process the “Azure” type element in this case. |

**Child elements of “*StorageType***”:

**Log**

Contains details about options for processing log dataset.

|  |  |  |
| --- | --- | --- |
| **Attribute Name** | **Values** | **Description** |
| LogDir | A path name | Name of the directory where the logs are stored |
| FilePrefix | A string | Only process log files starting with the given prefix. |
| LogWriteInterval | Numeric value | The counter log written interval in seconds |
| Iterations | Numeric value | The number of epochs to be taken for each counter analysis. When not present all the data will be processed. |

**Azure**

Contains details about the options for processing performance counter data from Azure storage

|  |  |  |
| --- | --- | --- |
| **Attribute Name** | **Values** | **Description** |
| ConnectionString | String | The connection string to connect to the storage. |
| TableName | String | The name of the NOSQL table where the data is stored. |
| PartitionKey | String | The partition key of the dataset. |
| LogWriteInterval | Numeric value | The counter log written interval in seconds |

In the case of Azure it has to read all the data. Reading certain number iterations is not yet supported in Azure storage based data collection.

**2. UserAnalysis**

This is for user driven analysis. User can specify set of counters and analysis methods.

It has the following child elements.

**Child elements of “UserAnalysis”**

**Counters**

This allows user to provide set of counters for analysis using the tool. Each counter should be specified as inner text using the <Name> element.

e.g <Name>Runtime.CPU.Usage</Name>

**Actions**

This element is used to specify the set of actions the tool will take to support the analysis of the given performance counter set. They are the ones described in Section 1. Following are the child elements, for each of these actions.

1. ***Visualization***

This is used to provide the Excel based visualization for performance counter data. Following are the attributes of this element

|  |  |  |
| --- | --- | --- |
| **Attribute Name** | **Values** | **Description** |
| ChartDir | A File path | The name of the location where the charts are generated. This location will be created if not exist. The excel file will be generated with the performance counter name. |
| Visible | True/False(Default) | Whether to show the Excel file generation process or not. |

1. ***ThresholdAnalysis***

This will be used for single counter threshold analysis. The threshold based analysis will be done only for elements specified in the RuleConfig file. Following are the attributes of this element.

|  |  |  |
| --- | --- | --- |
| **Attribute Name** | **Value** | **Description** |
| RuleConfigFile | A file path | Location of the file, which has rules for threshold analysis. |

The configuration options for Threshold analysis is described at the end of this document.

1. ***CorrelationAnalysis***

When this element is specified the tool will do correlation analysis. Following are the attributes of this element.

|  |  |  |
| --- | --- | --- |
| **Attribute Name** | **Value** | **Description** |
| Pearson | True/False(Default) | This will calculate Pearson coefficient for each KPI with all the explanatory counters and output the correlations which are greater than 0.9 |
| Spearman | True/False(Default) | This will calculate Spearman coefficient for each KPI with all the explanatory counters and output the correlations which are greater than 0.9 |
| HistogramExplanation | True/False(Default) | This will show the output of the correlation analysis with Histogram Explanation. |

For more details about correlation analysis techniques please see the Appendix below. It has a detailed explanation on correlation analysis, from Christian Konig. Specially about the histogram explanation technique.

1. ***VarianceAnalysis***

When this element is specified the tool will do the variance analysis. This is still in experimental stage. It will output the counters which have very high variance.

1. ***PerformanceTuning***

When this element is specified the tool will do performance tuning analysis. This is still in experimental stage. It will output comma-separated values which the user can save as .csv and read using Excel.

**3. DataAnalysis**

This is used for data driven analysis. When this is specified, tool will find suspicious counters and the silos where the values of the counters are anomalous. It will do all the analysis specified in the “Actions” child element for the suspicious counters it found. User does not need to specify any counters using “Counters” element in this case. When the tool is doing this analysis, the user has to provide a threshold value. It is an interactive process, user can provide different threshold values to get an idea about the outliers.

**ComparativeAnalysis**

This is used to compare all the datasets in the configuration file with respect to the reference dataset. This is similar to “AnomalyDetection” but now the suspicious counters will be reported with respect to the reference dataset. This will compare global statistics of the reference dataset and other datasets. To enable silo and time level comparisons following attributes need to be set to True.

This element has following attributes to control the number of counters reported. Similar to data analysis process this is also an interactive process.

|  |  |  |
| --- | --- | --- |
| **Attribute Name** | **Values** | **Description** |
| Silo | True/False | Tells whether the counter statistics should be compared with the reference data set at silo level |
| Time | True/False | Tells whether the counter statistics should be compared with the reference data set at time level. |

Following section describes the threshold based analysis process. The user has to provide a separate configuration file specifying rules for the threshold based analysis.

**Threshold based analysis configuration elements.**

The threshold based analysis will be specified using rules. These rules must be defined in RuleConfigFile. Following is a description about the RuleConfigFile. “PerfromanceCounter” element is used to define the set of counters for which the rules need to add. It has the following attribute.

|  |  |  |
| --- | --- | --- |
| **Attribute Name** | **Value** | **Description** |
| Names | String | Names of the counters the rule will apply. This will be a comma separated list, if the rule is to be applied to multiple counters. If the rule needs to be applied to all the counters, please specify “All” |

All the rules are closed within a “Rules” parent element and each can have any number of rules. Each “Rule” element have the following attribute.

|  |  |  |
| --- | --- | --- |
| **Attribute Name** | **Value** | **Description** |
| AppliesTo | Global | The rules will be applied to the global data. |
| Silo | The rules will be applied to each silo across time. |
| Time | The rules will be applied to data of each time point across silo. |

Each rule can have following child element and inner text values.

|  |  |  |
| --- | --- | --- |
| **Element** | **Inner Text value** | **Description** |
| Name | String or an Id | Use to distinguish the rule from other rules. |
| Statistic | Average, Median, Max, Min, Sdev, Variance, Zscore, Any | The value “Any” will check all the row values in the dataset with the expected value. Value Zscore will check corresponding Zscore value of a particular row value with the expected value. Other values for statistic element are self-explainable. |
| ExpectedValue | Numerical value | The user expected value for the analysis |
| ExpectedPercentageValue | Numerical value | The user expected percentage difference with a given statistic |
| ComparisonOperator | LessThan | Check whether the expected value is Less than the current value of the counter |
|  | GreaterThan | Check whether the expected value is greater than the current value of the counter |
|  | EqualsTo | Checks whether the Expected value is equal to the current value. |

The threshold analysis will output all the counter value instances which breaks any of the specified rules, with the counter name.

**Appendix (Correlation analysis techniques a detailed description given by Christian Konig in XCG)**

Dear Jorgen, Gabriel & Thelge,

Please find below the follow-up items from my side from Friday’s meeting:

(1) Please find attached the “Machine Learning in Practice” presentation that we partially went over; if there’s interest, I’ll gladly present the entire talk (e.g., as part of your reading group).

(2) The ‘TLC’ tool that we discussed (essentially, a MS Machine-Learning Library) can be found at [\\sbgfile\tlc](file:///\\sbgfile\tlc) (the folder also includes demos, a tutorial and some very good documentation).

(3) Please find attached the code for the 3 correlation functions we had discussed last week, and a small test-script that shows how they are called. I’ll describe them in more detail below.

The code consists of two files – ‘StatisticalFunctions.cs’, which contains all the correlation functions and ‘Program.cs’, which contains a number of examples invoking them.

The easiest correlation functions to call are the two correlation coefficients – Pearson’s and Spearman’s; the corresponding calls are

double P\_correlation = SF.Pearsons(X, Y);

double S\_correlation = SF.Spearman(X, Y);

where X,Y are of type Lists<double>. Pearsons is an implementation of the Pearson’s ‘standard’ linear correlation measure and Spearman covers non-linear (= rank) correlation.

However, there are many scenarios where these types of correlation are not what you want in practice: they are hard to interpret by humans (“what does a correlation coefficient of 0.67 mean?”), and they have trouble with counters that are uncorrelated in ‘normal operations’ and only exhibit high correlation when problems arise (which is something I’ve seen happen repeatedly in the context of SQL queries – e.g., memory consumption and performance are not highly correlated, unless memory becomes scarce).

For this purpose, we use the ‘Histogram Explanations’ that I showed the Pillar talk; these again are a measure of the correlation between two counters, but here the output is not a single number, but a partitioning of the counters into k buckets. The typical case here is k=2, which can be thought of as the automated partitioning of counters into the “baseline” case and the “problem” case. Also, for ‘Histogram Explanations’ we treat the two counters in an asymmetric way; similar to the ‘general approach’ we discussed on the board, one is considered the ‘Counter of Interest’, i.e., the counter whose change we want to diagnose; the other counter is the ‘Explanation Counter’, i.e., the counter for which we want to test is it’s variation ‘explains’ the changes in the ‘Counter of Interest’.

The way this works internally is that the algorithm partitions the domain of the ‘Explanation Counter’ into buckets such that the variance within the buckets is minimized. Then, we use the same partitioning on the ‘Counter of Interest’ and compute the resulting in-bucket variance for the values of this counter. This variance then becomes the correlation ‘score’ by which we rank the different explanations.

To give a simple example how this works, consider the case where the ’Counter-of-Interest’ is the duration of a request and the ‘Explanation Counter’ is the amount of memory given to it by the scheduler; in this example, the requests take 10-20 seconds is they receive sufficient memory (which may range between 5-10 MB) and 50 seconds if they receive less than that. Now, if we have a system that is running out of memory over time, we might see a sequence such as this:

Counter-of-Interest (in sec.):                     19           10           11           16           50           50           50

Explanation Counter (in MB)                       5           10           7              10             1           3              2

Now, the domain of the explanation counter would be partitioned into 2 buckets, the first one containing values [1…4] and the 2nd one values [5...7], corresponding to the baseline (= sufficient memory) and the ‘problem’ case (= too little memory). Now, the variance of the ‘Counter of Interest’ in the 2nd buckets is 0 and in the 1st bucket is low, so this qualifies as a good explanation.

Now, by doing the same call for various counters related to the ‘Counter of Interest’, it’s easy to identify which ones are the best ‘explanation’ for the behavior of a given ‘Counter of Interest’.

To compute the ‘Histogram Explanation’, the call is:

HistogramExplanation  Explanation1 =  GenerateMinVarianceHistogram(List<object> COI, List<object> C\_Explain, int NumberOfBuckets, bool OrderByCOI, DependencyType Dependency, CorrelationType Correlation)

Here, COI and C\_Explain are lists of values for the ‘Counter of Interest’ and the ‘Explanation Counter’, NumberOfBuckets is the number of ‘buckets’ the histogram should partition things into (typically two) and Dependency and Correlation are used to encode additional information about the dependency between the counters; If you don’t want to encode any additional information, then the defaults here are DependencyType.Default and CorrelationType.Either. Otherwise, using CorrelationType.Positive or CorrelationType.Negative signal that the two counters are positively/negatively correlated; if the resulting explanation does not show this trend, then ‘null’ is returned instead. The different values for the DependencyType are used for the scoring of different explanations: DependencyType.Additive means that a change of +/- X in the Explanation Counter implies a change of +/- X in the Counter-of-Interest, whereas DependencyType.Linear means that a change of +/- X in the Explanation Counter implies a change of *at most* +/- X in the Counter-of-Interest.

Finally, the parameter OrderByCOI  is used to differentiate between two types of analysis: if it is set to false, everything proceeds as explained above; if it is set to true, the sequence of input counter values is sorted, by increasing value of COI.  This corresponds to a different type of analysis: in the case of OrderByCOI=false, the histograms explain changes *over time*, whereas in case OrderByCOI=true it explains the *variance* in the counter of interest – for example, if you  took the example described above, but randomly shuffled the measurements (i.e., instead of the system state first having enough memory and then being memory starved, the memory availability fluctuates randomly over time), you would likely get the exact same explanation as before when OrderByCOI is set to true. In practice, I’ve found setting OrderByCOI=true works best in most practical scenarios.

If you’re interested in how this works precisely and when to use which parameter, just let me know and I can sketch it on the board.

Finally, the code also works when the Explanation Counter is non-numeric (i.e., categorical); the attached code contains an example (“Example 2”) where we correlate a numeric counter of interest to a set of string values. In this case, I require that OrderByCOI=true.

The histogram explanation itself uses this data structure: I hope the comments below are sufficient…

    public class HistogramExplanation

    {

        public int NumberOfBuckets { get; private set; }            // The number of different boundaries the domain of the counter is partitioned in

        public List<int> Bucketboudaries { get; private set; }      // Records the offsets of the bucket boundaries

        public List<double> COI\_Aggregate { get; private set; }     // Records the aggregate (sum/average) of the COI within each bucket

        public List<double> CE\_Aggregate { get; private set; }      // Records the aggregate (sum/average) of the C\_Explain within each bucket

        public List<List<object>> CE\_Values { get; private set; }   // Contains the individual values for the CE in case it's categorical

        public double Score { get; private set; }                   // The score used for ranking

        public bool VarianceAnalysis { get; private set; }          // Does the order of the tuples reflect time or the size of the COI.

        public int NumberOfObservations { get; private set; }       // The total number of observations this histogram is built over

        public bool CE\_is\_Categorical { get; private set; }         // If this value is true, then the CE attribute is categorical and

…

**Performance Considerations:**Since the data sets I used when monitoring SQL Server Counters where compact (few 1000s of ‘measurements’ for each counter) and I wanted to keep the code easy to understand, the code is not much optimized for performance; still, the two correlation measures should already be very fast and I know how to speed up the Histogram generation as well, but haven’t implemented this in the version I sent. So if things are too slow for your purposes.

**Visualizing the output:** The code currently has a simple .Print() function to output the Histogram Explanations – if you’d like to use the visualization used in the demo instead, we can also look into the corresponding WPF code.

I hope this should be enough to get you started. Please feel free to ping me with any questions, etc.