Efficient Bottleneck Detection in Stream Process System Using Fuzzy Logic Model

Yanlong Zhai, Wu Xu
Beijing Engineering Research Center of Massive Language Information Processing and Cloud Computing Application
School of Computer Science, Beijing Institute of Technology, Beijing, China
Email: {ylzhai; wuxu}@bit.edu.cn

***Abstract*—Big Data has shown lots of potential in numerous domain and becomes one of the emerging technologies that are bringing revolution in some real world industry. It has the power to provide insights into the unseen aspects of immense volume of data. Some applications are processing the data using a store-then-process paradigm, whereas other applications, like telecommunications and large-scale sensor networks, have to analyze continuous data flow online. Stream Processing Engines(SPEs) are designed to support applications which require timely analysis of high volume data streams. The dynamic nature of data stream requires SPEs to have high scalability. However, current SPEs mostly adopt a static configuration and can not scale out/in flexibly along with the changing of the data stream. In this paper, we proposed a fuzzy logic based runtime bottleneck operator detection approach to improve the scalability of SPEs by providing resources in the cloud environment. Our experimental results show that the fuzzy logic component developed in this work could detect bottleneck operators efficiently. Compared with other bottleneck detection methods, the decision results generated by our approach is more flexible and will not scale out/in the system when the workload change instantly.**

***Keywords*—Stream Processing System; Fuzzy Logic; Scalability; Big Data;**

I. Introduction

Processing data on-the-fly is a significant feature of stream process systems, which makes it different from store-then-process off-line big data processing systems. There are quite a few big data applications that generate massive amounts of data continuously, like large-scale wireless sensor network(WSN) and video surveillance systems. It is not cost-efficient nor profit-efficient to store all generated data in the data center for later processing. Besides, the data might be valuable only within a short period of time (minutes or hours) after being generated. Unlike Hadoop, which is widely used in off-line data processing applications, Stream Processing Engines(SPEs), such as Storm [1], Apache S4 [2], Spark Streaming [3] and Apache Flink [4], are designed to process the stream data using a serial of operators without storing data in the storage system. Nowadays, lots of data-intensive applications and technologies require online massive data processing with minimal delay. For instance, social networking sites(SNS) like Facebook, Twitter and Weibo need to execute real time data mining query to analyze web logs; online marketplace service and payment service providers such as Paypal and Alibaba need to run sophisticated fraud detection algorithms on real-time.

To meet the requirement of high computation power and adapt to the dynamism of the stream, SPEs are often deployed in the Cloud environment to achieve better scalability. Cloud computing service provides “pay-as-you-go” model for users to better utilize computing resources. To benefit from the “pay-as-you-go” model, stream processing system should scale out on demand. There are two broad categories to scalability: horizontal and vertical scaling. Scale Horizontally means add more (or remove) nodes to a system, such as applying additional virtual machines(VMs) when the workload increasing and releasing virtual machines when the workload is decreased to a certain extent. Scale Horizontally also known as scale out/in. System architects can configure hundreds of computers as a cluster to aggregate computing power which is much more powerful than computers based on a single traditional processor. Scale vertically, also known as scale up/down, means to add resource to, or remove from, a single node in a system. Resources involving CPUs, memory or network bandwidth have to reconfigured at a running system. Vertically scale behaves as more resource provides more computing power to handle requests. Horizontally scale is much more simple, more robust and easier to achieve than vertically scale for vertically scale needs add resource to (or remove from) at run-time which needs operating system and virtualization technology support. Most Cloud computing platform provides horizontally scale when referring to scalability. In this paper, we focused on horizontally scale.

However, most of current stream processing engines often show an insufficient scalability as the architecture is based on static configurations. Although some research and industry effort has been invested on scale out of operators in SPEs, most of them focus on how to scale out different type of operators based on an on-demand infrastructure. Few of them consider when and which operators should be scaled out, as improper scale out may introduce extra overhead.

Our previous work proposed a time utility function(TUF) based approach to find out the bottleneck operator at run time and scale out it only [8]. This time-aware utility accrual approach can exactly identify and efficiently scale out the bottleneck operator at run time in data stream processing system. Considering threshold-based approach may encounter “jitter” problem when the workload is continuously changing around the threshold, a more flexible approach is required to detect bottleneck operator and scale out/in the system.

2377-5750/17 $31.00 © 2017 IEEE
DOI 10.1109/PDP.2017.71

In this paper, we present a fuzzy logic based runtime bottleneck inspection and flexible system scale out/in approach. A comprehensive system model is defined to describe the stream, operator and operator scale out/in. The workload is also measured and modelled by CPU utilization, memory usage and data tuple size. This approach is stimulated by the observation that fuzzy logic is widely used in flexible industrial control field and some Non Player Character's(NPC) decision-making in video games. We fully evaluate our approach in a stream process system with generated simulation stream data.

The rest of this paper is organized as follows. Section II describes the models we used in the paper and formulate the problem formally. The detailed description of our approach presented in Section III. Section IV gives a integration design with storm and section V provides the evaluation of our approach using a stream processing system. Related works are discussed in Section VI and we present the conclusions in Section VII.

II. Problem Modeling

A. Problem Statement

We deploy stream process systems to infrastructure-as-a-service (IaaS) cloud platforms, such as Amazon EC2 and Microsoft Azure, across tens or hundreds of virtual machines. Cloud platform makes it possible to automatically deploy new prepared instance for scaling out very quickly. Also Cloud platform supports *pay-as-you-go* payment that makes dynamic scale gain economic benefits. Scale-in in low workload means deallocate virtual machines and saving money. Scale-out only when it's needed means efficient use of pre-allocated virtual machines.

A data stream is potentially an infinite sequence of tuples following a specific schema. In common stream process scenarios, the load of the stream is not consistent but changes a lot. For example, SNS system process much more data from 11:30 am to 1:30 pm than other time of day. Because most users prefer to read and tweeting during the midday-rest. When it comes to midnight, few users post tweets. That means we can unload many nodes to save money.

B. Stream Model

Data stream S is an infinite sequence of tuples t (t$\in$S), donated as S = (t_1, t_2,..., t_n). Tuples following the same schema: t =(τ, κ, γ, ρ). A tuple has a logical timestamp τ, a key field κ and an arrive time filed γ, while ρ represents the tuple size or the data content of the tuple. The timestamp is assigned by a monotonously increasing *logical clock* when the tuple is created. Keys of the tuples are not required to be unique for it is used to partition tuples into different operators but not to identify it from other tuples. The key value can be generated as a hash code of the tuple. ρ can be the size or the payload of the tuple [8], or just the content of the tuple.

Processing tuples leads to Operator performance's ups and downs. Generally speaking, Tuple needs more computing power when its size is bigger or its payload is higher. In other words, more complicated tuple cause the resource utilization raising when smaller one lowers it.

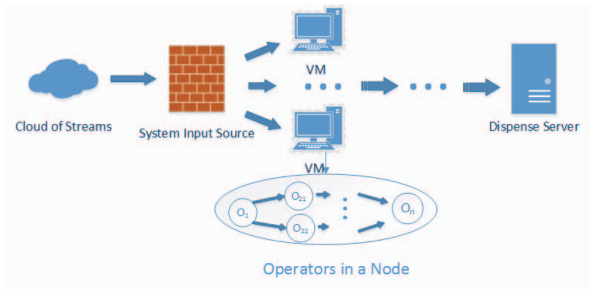


Fig. 1: Relationship between Nodes, operators and Query

C. Operator and Query Model

O = $\{O_1, O_2, \ldots, O_n\}$ is the set of operators. In SPEs, tuples are Processed by operators. And Operator o takes n (n$\geq$1) input streams, denoted by I_o = $\{s_1, \ldots, s_n\}$. Operator processes these stream's tuples and produces one or more output streams, usually only one stream O_o produced and we assume that an operator emits only one output stream.

A query in stream process is specified as a directed acyclic *query graph* [7]. Query graph consists of a set of operators and a set of streams denoted as $q = (O, S)$. A stream s $\in$ S is a directed edge connecting two operators, $s = (o, o')$ and $\{o, o'\} \subseteq O$. For an operator u is upstream to operator o, It is denoted as $u \in up(o)$ when $\exists(u, o) \in S$.

For the architecture of most stream processing systems, a query is deployed on a set of nodes. A node can host multiple operators, as shown in Figure. 1, but without loss of generality and to simplified model purpose, we assume that one operator holds only one node. We distinguish between the logical representation of a query, in terms of its query graph, and its physical realisation, as shown at the bottom of Figure. **??**. In the physical execution graph q, an operator o may be parallelised into a set of partitioned operators $o^1 \ldots o^\theta$. The value $\theta \in \mathbb{N}^+$ is the parallelisation level of o. Each o^i implements the semantics of o.It takes as input a partitioned stream $s^1 \ldots s^\theta$, which is obtained from the original stream s. We assume that one node split into only two nodes in the scale-out to simplify the model.

Figure. 1 shows the queries of a stream processing system and the simplified structure of stream processing system.

D. Scale Model

A query is deployed on a set of *nodes*. We use node represents a single virtual machine and a single node can host λ ($\lambda \geq 1$) operators. As mentioned above, a node hosts λ operators and always use one as the source of these operators and one operator can process *n* streams. A burst of tuples density of streams causes the raising of system resource utilization (such as CPU and memory utilization) and need to split the streams into two nodes to avoid the failure of the old node.

Castro Fernandez and Raul, et al [7] provides a backup-and-restore scale out algorithm. We can extend it to support both scale-out and scale-in easily. The goal of this paper is to present a novel approach to find out the bottleneck node of the system, rather than how to scale out a node, so we just give a brief description about the mechanism of scale-out and -in and [7] gives more detail about it.

III. Fuzzy-Logic Model Design

There are already some studies in the field of bottleneck detection. Mahammad Humayoo et al. [8] proposed an algorithm using Time Utility Function (TFU) model for operator scale out. Roy, Nilabja et al. [12] developed a model-predictive algorithm for workload forecasting used for resource auto scaling. Dutreilh Xaviver et al. [13] and E Barrett [10] both introduced a model that integrating the reinforcement learning algorithm into a real cloud controller for autonomic resource allocation. M. Lee [9] proposed a distributed and load adaption techniques for explosive data stream. From a performance point of view, the Time Utility Function method need to run an integral transform continually, that needs much extra computing power overhead especially for an already overloaded node. For reinforcement learning algorithm [10], the results is heavily rely upon the training data and can lead to a really negative training result, and the performance at initial stage can be very bad.

Our key idea is to use fuzzy logic control theory to estimate the status of a node and make the scale-out or scale-in decision. The remainder of this section will introduce the detail of our approach.

A. *Fuzzy Logic Theory*

Different from traditional Boolean logic that the truth value of variables may only be 0 (false) or 1 (true), fuzzy logic is a form of *many-valued* logic in which the truth value can be any number between 0 and 1. Fuzzy logic is an extension of boolean logic to handle the concept of partial truth scenario. The value of fuzzy logic variables represents the degree of truth which is different from the probabilities though they may seem similar. Fuzzy logic uses degrees of truth as a mathematical model of vagueness, while probability is a mathematical model of ignorance.

As mentioned previously, Fuzzy logic systems are used in industrial control domains. Advantages of Fuzzy logic control system lists as follows:

$\ast$ It can simplify the design of the system. It suits the system that is nonlinear or time-variable.

$\ast$ It uses control laws to describe the relationships between system parameters.

$\ast$ Using linguistic variables rather than numerical values make it possible that the Controller don't have to build a complete mathematical model.

The Fuzzy Logic Process has 3 steps: (i) Fuzzify all input values into fuzzy membership function. (ii) Execute all applicable rules in the rule set to compute the fuzzy output function. (iii) Defuzzify the fuzzy output functions to get "crisp" output values. The following sections will introduce how we realize them.

B. *Input and Output Variables*

With the purpose of simplifying the model, we assume that one node process only one stream at a time. In order to figure out the node's status, we focus on four input variables mainly, as follows:

- $c_i(t)$, the current CPU utilization (measured in %)
- $m_i(t)$, instantaneous Memory utilization (measured in %)
- $s_i(t)$, size of the tuple being processed (measured in KB)
- $miss_i(t)$, count of missed tuples that timeout before process (measured in number, optional)

While *i* denotes the stream *i*, *t* is the system logic timestamp. Stream process system can get the value of system's CPU and memory utilization thought programming interface. The size is one of the tuple's properties. As for the optional *miss* variable, we assume that SPEs is allowed to sustain losses of 0-10 tuples (timed out) just passing by without process. This is reasonable for some applications in real word for they are not so severely constrained. For example, a telecommunication fraud detection application has to return the result in a very short time, the stream may burst into 10,000 queries/min and when 3 of them are just timed-out, we still think this node works fine. Because the scenario is not so exigent and this lost is acceptable when compared with the cost of scale-out a node. But if one node keeps lost queries (time-out), it's time to consider the scale-out of this node. we have ever token disk performance as a separate variable, but traditional hard disk driver can perform tens of mega-bytes per second. The distributed file system can handle the write request. The input variables denoted as follow:

$$In_t = [c_i(t), m_i(t), s_i(t), miss_i(t)]$$

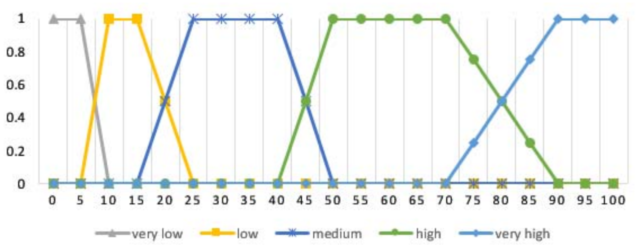
The output values of the fuzzy logic system is a number, and for calculate and understandable reasons, we set the range of output between 0 and 100, the the output can be map to a set of actions, that is

$$Out = \{scale\_out, keep, scale\_in\}$$

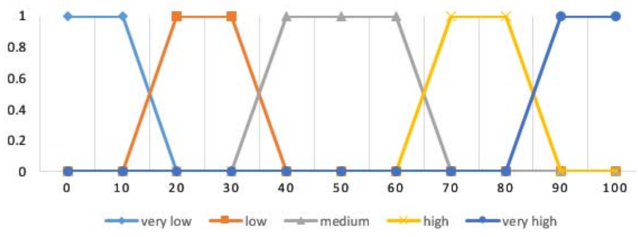
C. *Fuzzification*

Fuzzification operations can map mathematical input values into fuzzy membership functions [14]. As preparatory works of fuzzification, setting linguistic variables and membership function is a crucial job. Linguistic variables come from people's daily life. For example, we may believe CPU utilization is high when it comes up to 70%, but some others may think it is very high when some consider 70% is not so above the average, so it's just a normal utilization. This is the degree of truth we mentioned in section fuzzy logic theory.

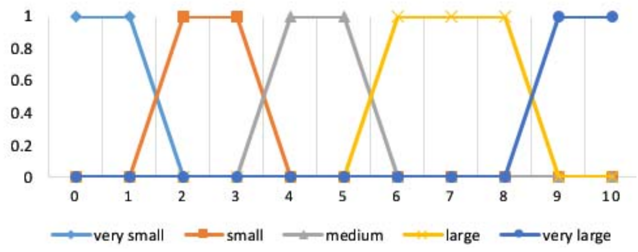
Each input variable is fuzzified using the fuzzy sets and membership functions shown in Fig. 2. We use the *jFuzzyLogic* Java library [15] to implement the Fuzzy System. As an example, For CPU utilization, we consider five linguistic labels, i.e, *very low(vl)*, *low(l)*, *medium(m)*, *high(h)* and *very high(vh)*.



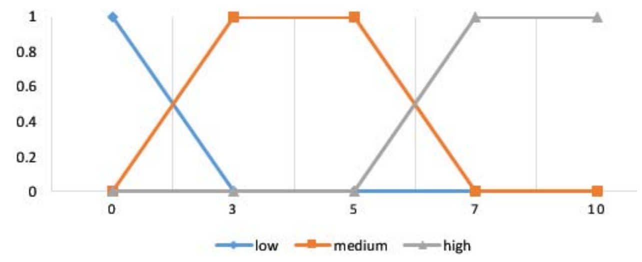
(a) CPU Utilization



(b) Memory Utilization



(c) Size of current tuple



(d) Missed tuples count

Fig. 2: Fuzzification(Fuzzy set) of Input variables

Memory, Size are divided into 5 labels as CPU utilization. We take three labels for missed-count as low/medium/high for the range of it is much smaller. It is worth mentioning that Memory utilization value is a bit different from CPU. Most of IaaS platform are built with Linux operating system, and Linux always tries to use RAM to speed up disk operations by using available memory for buffers (file system metadata) and cache (pages with actual contents of files or block devices). We have to take this into consideration when calculating memory utilization.

The output variable will be classified into three categories: *scale_in*, *keep* and *scale_out* (see Fig. 3). The values (or score) of output vary from 0 to 100, and smaller values stand for

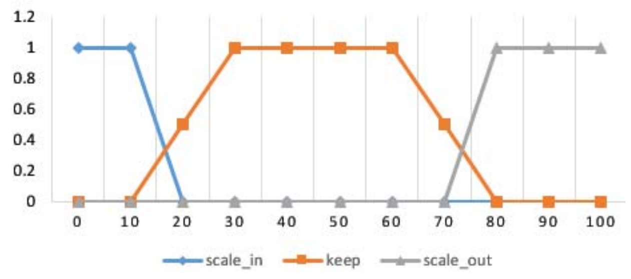


Fig. 3: Output variable: Scale Action

scale_in while bigger for *scale_out*.

D. Fuzzy Rule Set and Inference

Every input variable of the fuzzy logic system can belong to more than one linguistic label. For example, the CPU utilization can be *medium* and *high* at the same time. It means, generally, one set of input will match more than one rule.

A typical fuzzy logic rule looks like: $if\ x\ is\ A\ then\ y\ is\ B;$. As for our application, It's some more complex for its multi-dimensional.

$$if\ c\ is\ A\ and\ m\ is\ B\ and\ ...\ then\ output\ is\ K$$

A, B and K belong to corresponding fuzzy sets defined above. A subset of fuzzy rules of our rulebase is shown in Table I. These rules are generated from daily experience and improved by our evaluation results.For example, when cpu and memory utilization are both higher than 80% and mapped to fuzzy labels as very-high, we directly thinks the node is overloaded and need scale-out. Each individual rule combines different values for the input variables and output one value among the three available actions (i.e, $scale\_out$, $keep$, $scale\_in$).

TABLE I: Fuzzy inference rules

No	Rule
1	(c=vh) & (m=vh) ⇒ scale_out WITH 1
2	(c=vh) || (c=h)) & (m=vh) ⇒ scale_out WITH 0.8
3	(c=h) & ((m=vh) || (m=h)) ⇒ scale_out WITH 0.8
4	(c=h) & (m=h) & (s=vh) ⇒ scale_out WITH 0.8
5	(c=h) & (m=h) & (s=h) ⇒ scale_out WITH 0.7
6	(c=m) & (m=vh) ⇒ scale_out WITH 0.8
7	(c=vh) & (m=m) ⇒ scale_out WITH 0.6
8	(c=vh) & (m=l || m=vl) ⇒ scale_out WITH 0.5
11	(c=m) & (m=h) & (s=h || s=vl) ⇒ scale_out WITH 0.4
12	(c=vl) & (m=vl) & (s=vl) & (miss=l) ⇒ scale_in WITH 1
13	(c=vl) & (m=vl) & (s=l || s=m) & (miss=l) ⇒ scale_in
14	(c=vl) & (m=vl) & (s=vl) & (miss=m) ⇒ scale_in WITH 0.8
18	(c=l) & (m=l) & (s=l) & (miss=l) ⇒ scale_in WITH 0.6
19	(c=m) & (m=vl) & (s=vl) & (miss=l) ⇒ scale_in WITH 0.5
20	(c=vl) & (m=m) & (s=vl) & (miss=l) ⇒ scale_in WITH 0.5
21	(c=m || c=l) & (m=m || m=l) & (miss=l || miss=m) ⇒ keep
22	(c=h) & ((m=m) || (m=l)) & ((miss=m) || (miss=l)) ⇒ keep
23	(c=m || c=l) & (m=h) & ((miss=m) || (miss=l)) ⇒ keep

Weightings can be optionally added to each rule and weightings can be used to regulate the degree to which a rule affects the output values. We use static rule weightings based upon the priority or importance of each rule.

Each fuzzy rule ($if\ x\ is\ A\ then\ y\ is\ B$) can be regarded as a map from fuzzy set $A \in F(x)$ to fuzzy set $B \in F(y)$. There is a fuzzy transformation:

$$T : F(x) \to F(y)$$

which makes $T(A) = B$ when $A \in F(x)$ and $B \in F(y)$. From the fuzzy rule between X and Y, we get a fuzzy transformation. In other words, if $R \in F(X * Y)$, then R determines a transformation $T_R : F(x) \to F(Y)$. That is:

$$\forall A \in F(X) \ \Rightarrow\ T_R(A) = A \circ R \in F(Y)$$

we can use this formula to calculate fuzzy rules. But in our application, it's more complex because it has four input variables. There are two ways to handle the multidimensional inference. (i) Tsukamao Fuzzy Models [16]. (ii). Sugeno Fuzzy Models [16]. The Sugeno Fuzzy Models (also known as the TSK fuzzy model) use a function in the consequent. Usually $f(x, y)$ is a polynomial in the variables x and y, but it can be any function that can describe the output specified by the antecedent of the rules. In our fuzzy system, there are four antecedents. The *ith* rule can be denoted as:

$$\begin{aligned} R^i &: if\ c\ is\ c^i,\ m\ is\ m^i, s\ is\ s^i, miss\ is\ miss^i \\ &then\ out^i(x) = c_0^i + c_1^i c + c_2^i m + c_3^i s + c_4^i miss \end{aligned}$$

This is a first-order Sugeno fuzzy model for $f(x, y)$ of consequent is a first-order polynomial. The output of a first-order TSK Fuzzy logic system, $O_{TSK,1}(x)$, can be computed by combining all *M* rules' output:

$$\begin{aligned} O_{TSK,1}(x) &\equiv \frac{\sum\limits_{i=1}^{M} f^i(x) O^i(x)}{\sum\limits_{i=1}^{M} f^i(x)} \\ &\equiv \frac{\sum\limits_{i=1}^{M} f^i(x)(c_0 + c_1 c^i + c_2 m^i + c_3 s^i + c_4 miss^i)}{\sum\limits_{i=1}^{M} f^i(x)} \end{aligned}$$

$f^i(x)$ in this equation represents the Ignition level (weighting) of the *i*th rule.

E. Defuzzification

Defuzzification is the process of producing a quantifiable result in fuzzy logic. The defuzzification process we consider in our application is the COG algorithm, which calculates the center of gravity of the output signal obtained after the evaluation of the different rules [17].

The output of defuzzification is a number range from 0 to 100. We divide it into three sections corresponding to scale actions defined before by imposing two thresholds (δ_1, δ_2), i.e. a node can be chosen for scale out if $O(x) > \delta_2$, or scale in when $O(x) < \delta_1$ and keep unchanged in other cases.

IV. Integration Design

We can integrate our implementation with Apache Storm, which is the most popular distributed stream processing system for now.

A. Apache Storm Overview

Apache Storm is a free and open source distributed realtime computation system. Storm makes it easy to reliably process unbounded streams of data, doing for realtime processing what Hadoop did for batch processing [1]. There are five key abstractions in storm system:

- Tuples – an ordered list of elements. For example, a tuple described at Stream Model chapter.
- Streams – an unbounded sequence of tuples.
- Spouts – sources of streams in a computation (e.g. a Twitter API)
- Bolts – process input streams and produce output streams. They can: run functions; filter, aggregate, or join data; or talk to databases.
- Topologies – the overall calculation, represented visually as a network of spouts and bolts (as in the following diagram)

A programmer codes an application as Storm topology of operators, sources and sinks, which is typically a directed acyclic graph. The topology specifies the bolts, the connections, and how many worker processes to use. The bolt, working as the operator of our model, consumes input stream from its parent spouts or bolts, does calculation on received data, and emit new data steam to downstream bolts.

Apache Storm has two types of nodes (or machines), Nimbus and Supervisor. Nimbus is the master node of the system and is responsible for scheduling tasks among workers. Nimbus analysis the topology and gathers the tasks to be executed and distributes the tasks to an available Supervisor. A supervisor has one or more worker process and it will delegate the tasks to worker processes. Worker process will spawn many executors to run the task. Each source(spout) and operator (bolt) in a storm topology can be parallelized to improve the throughput.

Bolts are stateless in Storm. In fact Storm is stateless in nature, this helps Storm to process real-time data in the best performance. The statelessness feature just meets the assumption of our stream model described in section II-B. But Storm is not entirely a stateless system. It stores its state in Apache Zookeeper [5].

B. Integration with Storm

Storm provides the ability of Pluggable scheduler, that means a user can implement his own scheduler to replace the default scheduler to assign executors to workers. Our implementation just runs as a custom scheduler in a Java class that implements a predefined IScheduler interface in Storm. We configure the class to use the "storm.scheduler" config in storm. [22]. The default scheduler of storm uses a round robin allocation to balance out load, and this may result in tasks being placed at different workers. The re-balance operation of

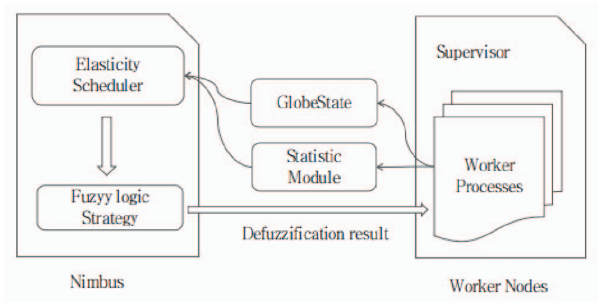


Fig. 4: Structure of Integration

storm for now is simply deleting the current scheduling and reschedules all tasks from scratch in a round robin fashion to the modified cluster.

Xu Le, Boyang Peng et al [21] has introduced a integration approach in storm by using customized scheduler in Stela. Its key point is to run a custom scheduler as part of the Storm Nimbus daemon. And it uses three modules to implement the integration: StatisticServer to collect statistics in the Storm cluster, GlobalState to store the important state information and Strategy module to provide an interface for scale-out strategies. Our work can integrate with Stale by customizing these three modules. StatisticServer still works on collecting information including cpu & memory utilization and throughtput of each node. We modify the origin strategy module to a fuzzy-logic-based one to calculate the schedule based on the model introduced in Section III and uses information from the Statistics and GlobalState modules. Stela runs a Elasticity Scheduler which starts the StatisticServer and GlobalState modules, and invokes the strategy module when needed. Figure. 4 shows the customized structure of the integration of our fuzzy logic model with storm based on the work of Stale. The algorithm of the strategy module to check if a node is overloaded described as Algorithm. 1.

C. Virtual Machine Pool

One More thing what is worth mentioning is when deploying a new node for scale-out, the time cost is a critical issue. VMs must be allocated as soon as possible, so it can set to work soon and resume the process. Current IaaS platform can provision new VM instances much faster than traditional service provider, but from our experience it requires on the order of several minutes to get things done. That means the stream process system can't work forward for several minutes but just wait for virtual machine get prepared. We can use a pre-allocated virtual machine pool to solve this issue [7].

Virtual machine pool is a set of well-prepared but idle nodes at the size of ω. Maintaining such a pool incurs an extra financial cost, but it can save the time of allocating a new instance from IaaS platform and speed up the procedure of scale-out.

Algorithm 1 Run botteleneck detection thread using fuzzy logic in each Node

```
Initiation: Let the operators $o_i$, i = 1, ...,k to process incomming tuples, Let $FIS$ be the fuzzylogic engine. Let $Status$ represents the node's status.
for all nodes do
    Init FIS with jFuzzyLogic library
    FIS.load('...')          ▷ FIS load fuzzy rules from file.
    if $FIS\ init\ failed$ then
        throw Exception
    else
        fis.getFunctionBlock();
        loop
            if !(get Status from node) then
                throw Exception
            end if
            FIS.setVariable($Status$);
            $Result \Leftarrow$ FIS.evaluate()
            if $Resule \geq Scale\_out\_threshold$ then
                This Node should scale-out
            else if $Result \leq Scale\_in\_theshold$ then
                This Node should scale-in
            else
                Keep processing
            end if
        end loop
    end if
end for
```

V. Evaluation

In this section, we evaluate our bottleneck detection approach with a fuzzy logic library. First we provide details of the experiment setup, then a detailed presentation and discussion of the results.

A. Evaluate Setup

To evaluate the effect of the fuzzy logic system we introduced previously, we have to setup a stream data producer from which randomly generates data tuples from time to time at first. We use an open-source project, *Json-Data-Generator* (*JDG*) [20], to play the role of stream producer. *JDG* can generate a real-time stream of json data, and it is a robust offline generator and support for user-defined data structure and different types of data, e.g. numbers, timestamp, string and array. We can integrate the *JDG* with out main Java program easily for its working-on-offline feature. *JDG* uses two configurations, the first one to define the producers and workflows, the second one to define the detail of data structure and repeat rules. The producer is a output destination of the stream which can be a local file, console or a network location and the workflow represents a stream which connected with a workflow config file. We config one workflow named *scale* and one producer only for evaluation purpose. Workflow detail config shown as Table. II, we config the stream generating tuples in randomly order of the steps and in a random time interval during 0 to *timeBetweenRepeat* variable.

TABLE II: Random stream wrokflow config

Property	Value
eventFrequency	100
timeBetweenRepeat	50
varyRepeatFrequency	true
repeatWorkflow	true
stepRunMode	random

As for fuzzy control programming, we use *jFuzzyLogic* [15] as the kernel of our fuzzy logic system. jFuzzylogic is an open source project which implements fuzzy control language with Java programming language. It supports multiple membership functions, defuzzifiers, rule aggregation and rule connection operators. And it provides easy-to-use programming interfaces. We use a superset of the rules introduced in Table I as jFuzzyLogic's rule base.

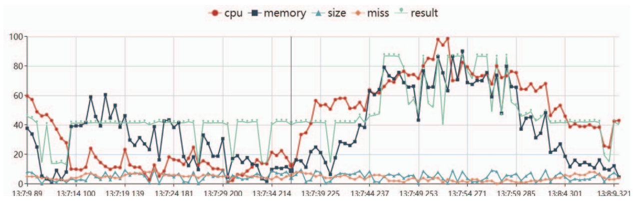
Park, Alfred J., et al [18] provides a application-level stream processing system simulator named *Flow*, and we build a similar one in Java to simulate the work of stream process engine. Each test suite consists of one Json-Data-Generator instance to create stream data, one stream process thread works as a SPE and one fuzzy logic inference thread to make scale strategy. The evaluate run as a Java application on an Linux virtual machine which has 6 logical processors and 6GB memory.

B. Results

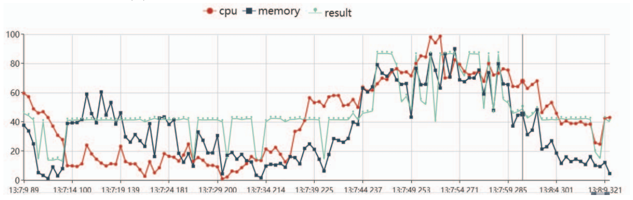
Fig. 5 and Fig. 6 give an overview of the results of our evaluation. It contains more than 120 continuous status-checking actions which is a small, but typical, subset of our full evaluation, with each dot presents one. The x-axis of Fig, 5 is a logical timestamp generated by the test platform and is monotonously increasing but have nothing to do with a real-world clock. Fig. 5a shows all four input variables and the output line together. The y-axis is the number of variables' value. For tuple-size and missed-count are both range from 0 to 10, they are displayed in the bottom of the figure.

Fig. 5b shows the strong correlation of CPU, memory utilization and output value. This meets the rules we defined in Table I that cpu and memory do play a more important role than other two input variables. We can find a positive correlation between output value and the cpu/memory utilization basically. In the middle of the figure, the output is much bigger than both cpu and memory value, and we will explain this phenomenon in the next paragraph. Fig. 5c shows the variation of the size of tuples and the count of missed tuples. Because tuples are generated by random, the size of tuples changes very quickly and irregularly. In general, the missed variables meet the changes of the size, and that just works as expected.

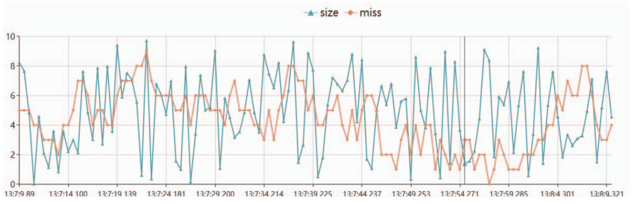
Fig. 6 shows the output line only. As we can see, There are several horizontal sections in the middle of the line which we mentioned in the previous paragraph. The values of these cases are around 40. The reason is that these input cases match no



(a) Overview of all variables of evaluation



(b) Strong correlation between cpu, memory and output



(c) Size and count of missed

Fig. 5: Results of evaluation

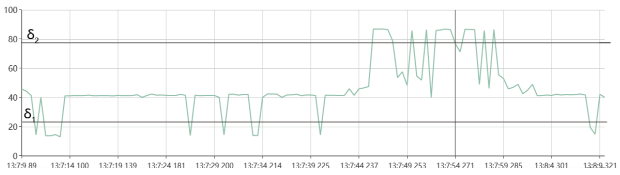


Fig. 6: Output of fuzzy system

fuzzy rule of our rulebase and the default value *40.12* specified by program was fetched as an output value. The cases whose value is bigger than δ_2 (*79*) should scale out and those whose value is smaller than δ_1 (*21*) should scale in.

Table. III shows the statistics of all cases' output. We have ran more than 5000 cases, the statistics shows that, for most inputs, the output of fuzzy logic system is to keep unchanged, and for about 7% of cases can picked to scale out, and for about 3% of cases is recommended to scale in. The statistic meets our expectation.

TABLE III: Statistics of the outputs of 5000 test cases

Action	**Count**	**Percentage**
Scale In	147	2.8%
Keep	4740	90.23%
Scale Out	366	6.97%
All	5253	100%

C. Summary of Evaluation

In this section, the evaluation shows that our fuzzy logic application can detect bottleneck nodes rightly and the result meets our expectation very well. In our application, the calculation is quite easy and with no integral computation or iterative learning process. We believe the performance of our algorithm is much better than other bottleneck detection methods, and we leave the evaluation of performance comparison to future work.

VI. Related Work

There exist some SPEs which provide both scale in and scale out or only one of them. Gulisano [6] describe how to partition the state of specific stateful operators such as join and aggregate for scale out. Each Stream Cloud periodically monitors incoming load and resource utilization to decide elasticity. They don't consider utility gain factor when the task is completed within its deadline and utility penalty when a task is aborted or not completed within its deadline.

Even in the absence of bottlenecks, if a VM hosting a stateful operator fails, the SPS must replace it with an operator on a new VM. [7] proposed an approach that overload and failure are handled in the same fashion. Operation recovery becomes a special case of scale out. Stela [21] presented a novel scale-out and scale-in techniques for stream processing systems with creating a novel metric, ETP (Effective Throughput Percentage), that accurately captures the importance of operators based on congestion and contribution to overall throughput. Stela also proposed an approach to integrate the scale component to storm and it is high available for our work.

Raphael Frank, et al. [19] use fuzzy logic system to process the data retrieved from vehicle's CAN bus by using an OBD adapter and provide the driver with a representative eco-score of their drive. The fuzzy system they used provides a representative eco-score based on three input variables and 24 inference rules.

The Flow stream processing system simulator [18] is a high performance, scalable simulator that automatically parallelizes chunks of the model space and incurs near zero synchronization overhead for stream application graphs that exhibit feed-forward behavior.

VII. Conclusion

In this paper, we presented an approach that can discover bottleneck nodes in stream processing applications. Our experimental results illustrate the effectiveness of our solution in finding an ideal bottleneck node for scaling out or scale in under present existing facts. This ability is attained by integrating fuzzy logic theory which commonly used in industrial control field with existing stream process engine. Because our algorithm just calculates over the node's key performance parameters and without integral computation or learning stages, it is quite steady and quick. As future work, we plan to extend our work to support node state management.

Acknowledgment

The research in this paper was supported by Grant 61602037 from the Natural Science Foundation of China. The authors thank the reviewers for their comments.

References

[1] "Apache Storm," https://storm.apache.org/, 2016, ONLINE

[2] Neumeyer, Leonardo, et al. *S4: Distributed stream computing platform.* Data Mining Workshops (ICDMW), 2010 IEEE International Conference on. IEEE, 2010.

[3] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters, in Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing. USENIX Association, 2012, pp. 1010

[4] Carbone P, Ewen S, Haridi S, Katsifodimos A, Markl V, Tzoumas K. Apache flink: Stream and batch processing in a single engine. IEEE Data Engineering Bulletin. 2015.

[5] "Apache Zookeeper", https://zookeeper.apache.org/, 2016, ONLINE

[6] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez and P. Valduriez, *StreamCloud: A Large Scale Data Streaming System*, Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on, Genova, 2010

[7] Castro Fernandez, Raul, et al. *Integrating scale out and fault tolerance in stream processing using operator state management*. Proceedings of the 2013 ACM SIGMOD international conference on Management of data. ACM, 2013.

[8] Humayoo, Mahammad, et al. *Operator Scale Out Using Time Utility Function in Big Data Stream Processing*. Wireless Algorithms, Systems, and Applications. Springer International Publishing, 2014.

[9] M. Lee, S. J. Hur, et al. *Load adaptive distributed stream processing system for explosive stream data*, 2015 17th International Conference on Advanced Communication Technology (ICACT), Seoul, 2015.

[10] E Barrett, E Howley, and J Duggan. *Applying reinforcement learning towards automating resource allocation and application scalability in the cloud*. Concurrency and Computation: Practice and Experience, 2012.

[11] Heinze, Thomas, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. "Auto-scaling techniques for elastic data stream processing." In Data Engineering Workshops (ICDEW), 2014 IEEE 30th International Conference on, pp. 296-302. IEEE, 2014.

[12] Roy, Nilabja, Abhishek Dubey, et al. *Efficient autoscaling in the cloud using predictive models for workload forecasting*. Cloud Computing (CLOUD), 2011 IEEE International Conference on. IEEE, 2011.

[13] Dutreilh, Xavier, et al. *Using reinforcement learning for autonomic resource allocation in clouds: towards a fully automated workflow*. Int. Conf. on Autonomic and Autonomous Systems. 2011.

[14] Klir, George, and Bo Yuan. *Fuzzy sets and fuzzy logic. Vol. 4*. New Jersey: Prentice hall, 1995.

[15] Cingolani, Pablo, Jesus Alcala-Fdez. *jFuzzyLogic: a robust and flexible Fuzzy-Logic inference system language implementation*. Fuzzy Systems (FUZZ-IEEE), 2012 IEEE International Conference on. IEEE, 2012.

[16] Jang, J. S. R., C. Sun, and E. Mizutani. *Fuzzy inference systems*. Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence (1997): 73-91.

[17] Mendel, Jerry M. *Fuzzy logic systems for engineering: a tutorial*. Proceedings of the IEEE 83.3 (1995): 345-377.

[18] Park, Alfred J., et al. "Flow: A stream processing system simulator." Principles of Advanced and Distributed Simulation (PADS), 2010 IEEE Workshop on. IEEE, 2010.

[19] Frank, Raphal, et al. "A novel eco-driving application to reduce energy consumption of electric vehicles." Connected Vehicles and Expo (ICCVE), 2013 International Conference on. IEEE, 2013.

[20] Json-Data-Generator, https://github.com/acesinc/json-data-generator

[21] Xu, Le, Boyang Peng, and Indranil Gupta. "Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand." IEEE International Conference on Cloud Engineering (IC2E). 2016.

[22] "Storm Scheduler," https://storm.apache.org/releases/1.0.1/Storm-Scheduler.html, 2016, ONLINE