2019 IEEE 35th International Conference on Data Engineering (ICDE)

# CEP-Wizard: Automatic Deployment of Distributed Complex Event Processing

Yooju Shin, Susik Yoon, Patara Trirat, and Jae-Gil Lee*
*Graduate School of Knowledge Service Engineering, KAIST*
Daejeon 34141, Republic of Korea
{yooju24, susikyoon, patara.t, jaegil}@kaist.ac.kr

***Abstract*—Complex event processing (CEP) is defined as event processing for multiple stream sources to infer events that suggest complicated circumstances. As the size of stream data becomes larger, CEP engines have been parallelized to take advantage of distributed computing. Typically, deployment of such a distributed CEP engine involves manual configuration, which has been regarded as an obstacle to its widespread adoption. In this demonstration, we present CEP-Wizard, a framework of automatically configuring and deploying a distributed CEP engine with minimum effort. The demonstration shows that even inexperienced users can easily configure and deploy it on Apache Storm with achieving high performance and low resource usage.**
**• Demo and Video: http://dmserver3.kaist.ac.kr/cep-wizard/**

## I. Introduction

### A. Motivation and Summary

Recently, complex event processing (CEP) has been widely used for various fields including Internet of things (IoT) and cyber-physical systems (CPS) [1]. A CEP engine receives multiple data streams from several sources and verifies whether they are matched against predefined *event rules* (or simply *rules*) [2]. As an example for vehicles, it monitors a rapid decrease of tire pressure from 45 psi to 20 psi within 5 seconds to detect a blowout. As the size of data streams, as well as the number of rules, keeps increasing, CEP engines are extended to run on a cluster of machines using a distributed stream processing system. We call such a kind of CEP engine a *distributed* CEP engine. When properly deployed, a distributed CEP engine definitely improves the performance of event detection from large-scale data streams.

However, deployment of a distributed CEP engine is complicated and burdensome simply because there are multiple machines and multiple rules. Each rule (e.g., a rapid decrease of tire pressure) needs to be handled by at least one machine. The optimal allocation is not straightforward because it needs to consider many factors such as the complexity of rules being monitored and the amount of data communications required. Users had to inspect their rules to decide how to allocate the rules optimally based on their intuition. Then, the users learned an application programming interface (API) to write the source code for a possibly unfamiliar system. These steps require users a lot of effort and time, so only experienced experts have usually conducted this job. This situation is not solved in most commercial distributed CEP engines, e.g., WSO2 [3]. On the other hand, in the relevant fields suffering from the similar situation, e.g., Map-Reduce, many efforts have been made to overcome the inefficiency of manual configuration and deployment [4]. These Map-Reduce solutions do not suit to the CEP environment because the data models in these two environments are completely different.

In this demonstration, we contend that the configuration and deployment of a distributed CEP engine should be *automatic* in order to improve the usability of a distributed CEP engine. Then, we demonstrate our on-going effort to develop an automatic deployment system of distributed CEP engines, which we call **CEP-Wizard**. The data flow of CEP-Wizard is shown in Figure 1, which consists of two main components: *rule allocator* and *code generator*. **(i)** The user provides a minimum amount of information including the data specification (stream schema and input rate), rule definition, and cluster specification (available resource). **(ii)** The *rule allocator* determines the optimal allocation of the rules to the machines in the cluster for efficient resource usage. The optimality condition mainly considers the amounts of stream data which should be accessed in all machines to monitor all rules. The resulting allocation is represented as a *stream rule graph* and provided to the next component. **(iii)** The *code generator* translates the stream rule graph to the executable code for a specific distributed stream processing system such as Apache Storm and Apache Flink.[1]

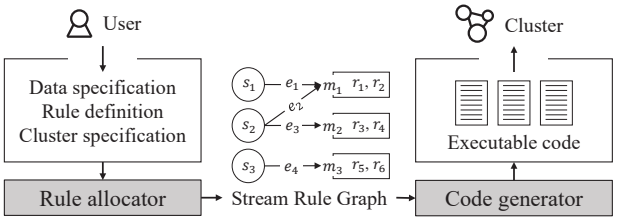



Fig. 1. Overall data flow chart of CEP-Wizard.

### B. Contributions

According to our design having two main components, our contributions are summarized as follows:

- **Rule allocator**: We develop a simple yet effective algorithm for allocating the rules to the machines. For

∗ Jae-Gil Lee is the corresponding author.

[1] At the time of this submission, the code generator is available for Apache Storm only. We plan to support other distributed CEP engines soon.

2375-026X/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDE.2019.00225

this purpose, we define the optimal condition of the allocation and present a greedy algorithm for finding the optimal solution. Then, based on the empirical evaluation using synthetic stream data sets, we confirm that our proposed algorithm achieves higher CEP efficiency than a baseline (round-robin) algorithm by 1.6 times.

- **Code generator**: We design CEP-Wizard to be *independent* from distributed stream processing systems. That is, it generates a neutral, intermediate data structure—a *stream rule graph*—which contains enough information to deploy a distributed CEP on *any* system. By this design principle, the developers can support other systems by merely adding the translators for them.

Regarding the optimal rule allocation, there has been insignificant research effort despite its importance. Kobayashi et al. [5] presented a rule allocation algorithm that tries to maximize the number of data streams shared by all rules in a specific machine. Pathack and Vaidehi [6] defined a point index of each rule to allocate similar rules together in a machine. However, these two studies did not consider the resource usage in the machine. Zygouras et al. [7] presented an algorithm to reduce the latency of each machine under the assumption of a *single* data stream with spatial attributes. Our proposed approach is free from these limitations.

## II. Architecture and Implementation

This section explains the details of the design and implementation of CEP-Wizard, following the order in Figure 1.

### A. Input Specification

CEP-Wizard requires three types of information. First, *data specification* includes a list of attributes with their name and data type as well as their input rate for each stream source. Second, *rule definition* focuses on a list of attributes accessed in each rule. The semantics of a rule follows the common convention of a CEP engine [8]. Third, *cluster specification* includes the number of available machines and that of available streams.[2] A short example is given in Figure 2.

```
{"Names":[
    {"1":"At0"},
    ...
],
"InputRates":[
    {"At0": "0.7"},
    ...
],...}
```

(a) Data specification.

```
Rule 1
  when
    At0 > 0.5
  then
    Turn on the alarm
Rule 2
...
```

(b) Rule definition.

```
{
"NumofMachines":5,
"NumofStreams":25
}
```

(c) Cluster specification.

Fig. 2. Example of the input to CEP-Wizard. (a) and (c) are given in the JSON format, and (b) is given in the Drools syntax [9].

### B. Rule Allocator

*1) Problem Definition:* Given a set of available machines $\mathcal{M} = \{m_1, \ldots, m_{N_m}\}$ and a set of defined rules $\mathcal{R} = \{r_1, \ldots, r_{N_r}\}$, the goal is to find the *optimal* set of allocations

[2]We assume that the performance of each machine is similar with each other. This assumption is reasonable since it is common to purchase or rent multiple machines of a similar specification.

**Algorithm 1** Greedy algorithm

**Input:** $\mathcal{R}, \mathcal{M}$, empty $\mathcal{A}$
**Output:** $SRG(\mathcal{A}^*)$

```
1: for i = 1 to N_m do
2:    rule ← Pop out a rule randomly from R;
3:    A ← A ∪ {⟨rule, m_i⟩};
4: for i = 1 to |R| − N_m do
5:    c ← Cost(A) /* Eq. (1) */, δ ← ∞;
6:    for j = 1 to N_m do
7:       if δ > Cost(A ∪ {⟨r_i, m_j⟩}) − c then
8:          δ ← Cost(A ∪ {⟨r_i, m_j⟩}) − c;
9:          j* ← j;
10:   A ← A ∪ {⟨r_i, m_j*⟩};
11: SRG ← StreamRuleGraph(A); /* ≈ A* */
12: return SRG;
```

$\mathcal{A}^* = \{\langle r, m\rangle \mid \forall r \in \mathcal{R}, \exists m \in \mathcal{M}\}$. Here, $\langle r, m\rangle$ means that a rule $r$ is executed on a machine $m$. Besides, each rule does *not* have dependency with other rules.

$\mathcal{A}^*$ minimizes the sum of the workloads on all machines, as follows. Let $\mathcal{R}_i \subseteq \mathcal{R}$ be a subset of the rules assigned to the $i$-th machine. $InputRate_i$ is the sum of the input rates of the streams required by $\mathcal{R}_i$, and $UnitCost_i$ is approximated as the total number of the conditions contained in $\mathcal{R}_i$. Here, as the number of conditions on a machine increases, its workload naturally increases to evaluate them. Thus, its workload is defined as $InputRate_i \times UnitCost_i$, and the objective function is the sum for all machines, as shown in Eq. (1).

$$Cost = \sum_{i=1}^{N_m} InputRate_i \times UnitCost_i \quad (1)$$

s.t. *Every rule is allocated to* **only one** *machine.*

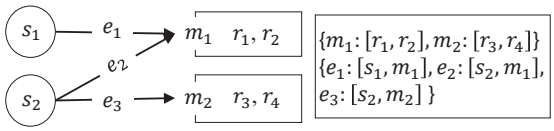
Although the strategy looks very simple, it can work better than Kobayashi et al. [5] and Pathack and Vaidehi [6], as described in Example 2.1.

*Example 2.1:* Suppose that there are two machines, three rules, $r_1$, $r_2$, $r_3$, and two streams, $s_1$, $s_2$ with input rate 1, 10, respectively. $r_1$ uses $s_1$ and $s_2$; $r_2$ uses $s_1$; and $r_3$ uses $s_2$. The existing algorithms cannot tell the difference between grouping $r_1$ and $r_2$ together and grouping $r_1$ and $r_3$ together, because they consider only the similarity of the rules allocated together. However, obviously, $r_1$ and $r_3$ had better be grouped together, because the input to these two rules has a high input rate. Our approach will choose this preferable option to reduce the duplication of data transmissions. □

*2) Greedy Algorithm:* Finding the optimal solution of Eq. (1) is NP-hard, because it is formulated by non-convex mixed integer nonlinear programming [10]. Thus, we adopt a greedy strategy, and Algorithm 1 shows its pseudocode. $N_m$ rules are randomly selected, and each rule becomes the first resident of each machine (Lines 1–3). Then, for each of the remaining rules, the algorithm allocates it to the machine that minimizes the increase of Eq. (1) (Lines 4–10). Last, the algorithm returns the resulting stream rule graph (Lines 11–12).

### C. Stream Rule Graph (SRG)

A *stream rule graph (SRG)* is a bipartite directed acyclic graph $\mathbb{G} = (\mathcal{S}, \mathcal{M}, \mathcal{E})$. Here, $\mathcal{S}$ is a set of stream sources, $\mathcal{M}$ is a set of available machines, and $\mathcal{E}$ is a set of data transmissions from stream sources $\mathcal{S}$ to machines $\mathcal{M}$. $m_i \in \mathcal{M}$ is associated with the attributes for a list of the rules allocated to $m_i$. This stream rule graph has basic information required for deploying a distributed CEP engine. In CEP-Wizard, it is also represented in the JSON format, as shown in Figure 3.




(a) Graphical illustration. (b) JSON representation.

Fig. 3. Representation of a stream rule graph.

### D. Code Generator

The code generator translates the stream rule graph into the executable code for a specific system. This translation should be easy because of the simple structure of the stream rule graph. For Apache Storm as an example, our code generator creates a *spout* for stream sources and a *bolt* on every machine. Then, it builds a *Storm topology* by connecting a spout and a bolt according to the set of edges.

## III. Experiment and Results

This section evaluates the effectiveness of our **rule allocator** component. To this end, we compare our **proposed** algorithm (Algorithm 1) with the **baseline** algorithm that allocates rules to machines in a round-robin fashion. A comparison with Kobayashi et al. [5] is omitted here, because it often showed poor performance even than the baseline algorithm for the reason in Example 2.1.

### A. Setting

To readily vary the characteristics of data and queries, we generate synthetic stream data sets and rules as follows. A *stream* $s_j$ is a univariate time series, $s_j = \{s_j^{t_0}, s_j^{t_1}, \ldots, s_j^{t_l}, \ldots\}$, where $s_j^{t_l}$ is the value of $s_j$ at the timestamp $t_l$ and follows $U(0, 1)$. The input rate (i.e., $\frac{1}{t_{l+1} - t_l}$) is assumed to be constant in the same stream and is determined to be a random value in $(0, 1)$. Then, a *rule* $r_k$ is generated using the following rule template. We fix the rule condition to be the conjunction of at most four arithmetic comparisons. That is, at most four streams ($s_{x_1} \sim s_{x_4}$) are randomly selected, and the corresponding thresholds ($V_1 \sim V_4$) are also sampled from the same uniform distribution.

*When*
$s_{x_1} > V_1$ && ... && $s_{x_4} > V_4$
*Then*
...

Since a comprehensive evaluation is beyond the scope of this demonstration, we present the results only for one of our configurations tested. A similar result was obtained for other configurations. We set the number of stream sources to be $|\mathcal{S}| = 25$, that of available machines to be $|\mathcal{M}| = 5$, and that of rules to be $|\mathcal{R}| = 300$. Each stream source generates the values until $20,000$ timestamps. We used five Amazon EC2 t2.medium instances to run a distributed CEP.

### B. Results

**Metrics**: First, *replication ratio* is the ratio of the total number of tuples processed to that of tuples generated. Second, *capacity* is an internal metric provided by Apache Storm, which indicates the ratio of the execution time of a CEP engine to the entire running time. The lower both metrics are, the better the performance is.

**Summary**: Figure 4 shows the performance results for the baseline and proposed algorithms. Our algorithm replicated data streams 33% less than the baseline algorithm, which means that the rules using the shared streams were effectively clustered in a small number of machines. At the same time, our algorithm lowered the average capacity of CEP engines by 32% compared with the baseline algorithm.

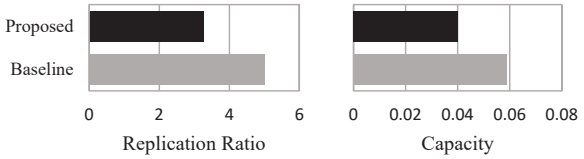



Fig. 4. Performance comparison between ours and baseline algorithm.

**Significance**: It is worthwhile to note that an inadequate rule allocation can cause the performance degradation of a distributed CEP engine. Our CEP-Wizard definitely facilitates this important design procedure with requiring only a small amount of information from users.

## IV. Demonstration Overview

This section describes the demonstration system and scenarios.[3] Figure 5 shows the graphical user interface (GUI) for our demonstration. Two scenarios are prepared at the time of this submission; the former uses an archived real-world data set, and the latter uses the synthetic data set in Section III.

**Overall Procedure**: Both scenarios follow the overall procedure in Figure 1. For comparison purposes, our demonstration system shows the results of both the proposed and baseline algorithms. **(i)** The input to CEP-Wizard is displayed like the upper-left part of Figure 5. **(ii)** Given the input, CEP-Wizard generates and displays a stream rule graph like the upper-center part. We can check which rules are in each machine by clicking the machine node. **(iii)** Next, CEP-Wizard generates and shows the executable code for a specific system, i.e., Apache Storm, like the upper-right part. **(iv)** Then, CEP-Wizard deploys distributed CEP engines and shows their status as in the lower-left part. **(v)** Last, CEP-Wizard fetches log files and displays the output of the rules like the lower-right part.

**Advantages**: We highlight the main advantages of CEP-Wizard using the real-world scenario. The data set consists of

[3]The demonstration system can be accessed from http://dmserver3.kaist.ac.kr/cep-wizard/, and a demo video is also available on the same page.

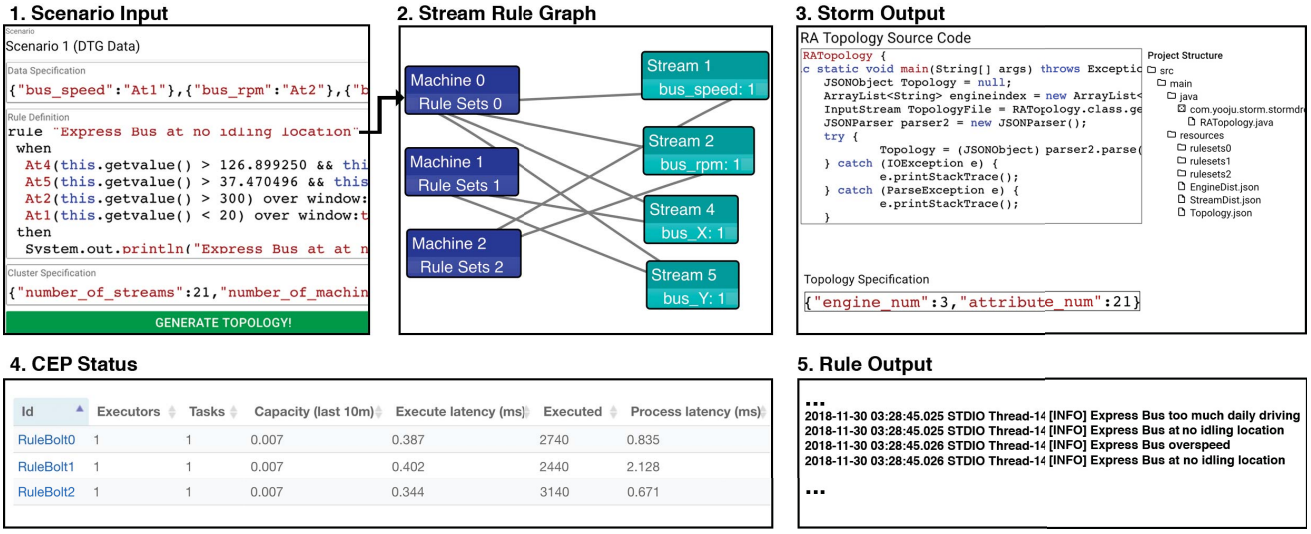



Fig. 5. The web interface of CEP-Wizard for demonstration (edited for readability).

21 streams collected from the sensors embedded in vehicles.[4] Each stream indicates the timestamp, daily mileage, speed, RPM, brake signal, location (latitude and longitude), etc. for three types of vehicles. 33 handcrafted rules are being monitored. One of the interesting rules is to detect idling at a prohibited area for express buses. As shown in the stream rule graph, CEP-Wizard allocated the rule to the machine which received the streams from the express bus sensors for the speed, RPM, longitude (X), and latitude (Y). These streams were essential to check the conditions in the rule, so the allocation was made correctly. Comparing the two stream rule graphs by the proposed and baseline algorithms (only one shown in Figure 5), our proposed algorithm established fewer edges than the baseline algorithm, thereby reducing the number of data transmissions. After a CEP engine was deployed using Apache Storm, the status panel showed that the workloads of the three machines (bolts) were well-balanced in that the *execute latency* of each machine, which is the average time a tuple spends in the execute method, all fell into the range of 0.3–0.4 *ms*. These results demonstrate the superiority of our rule allocation strategy. Last, the events detected by the 33 rules started appearing in the output panel. For example, the express buses were identified in real time when they were idling at a prohibited area, overspeeding, or drove too much in a day. Overall, all these steps took only a few minutes.

## V. Conclusion

In this demonstration, we introduced **CEP-Wizard**, a framework of automatically configuring and deploying a distributed CEP engine with minimum effort. The effectiveness of our automatic configuration (rule allocation) was empirically verified by the comparison with the round-robin algorithm. Through this demonstration, we confirmed that the deployment can be completed very quickly just in a few minutes. Therefore, we believe that our work will be very helpful to proliferate the use of distributed CEP engines.

[4]This data set was provided by the Ministry of Land, Infrastructure and Transport, Korea under a non-disclosure agreement.

## Acknowledgment

This research was supported by the MOLIT (The Ministry of Land, Infrastructure and Transport), Korea, under the national spatial information research program supervised by the KAIA (Korea Agency for Infrastructure Technology Advancement) (19NSIP-B081011-06).

## References

[1] C. Y. Chen, J. H. Fu, T. Sung, P. Wang, E. Jou, and M. Feng, "Complex event processing for the internet of things and its applications," in *Proc. 2014 IEEE Int'l Conf. on Automation Science and Engineering*, 2014, pp. 1144–1149.

[2] I. Yi, J.-G. Lee, and K.-Y. Whang, "APAM: Adaptive eager-lazy hybrid evaluation of event patterns for low latency," in *Proc. 25th ACM Int'l Conf. on Information and Knowledge Management*, 2016, pp. 2275–2280.

[3] T. Dahanayakage, S. Suhothayan, and M. Dayarathna, "WSO2 stream processor." [Online]. Available: https://wso2.com/analytics-and-stream-processing/

[4] M. B. S. Ahmad and A. Cheung, "Automatically leveraging MapReduce frameworks for data-intensive applications," in *Proc. 2018 ACM SIGMOD Int'l Conf. on Management of Data*, 2018, pp. 1205–1220.

[5] Y. Kobayashi, K. Isoyama, K. Kida, and H. Tagato, "A complex event processing for large-scale M2M services and its performance evaluations," in *Proc. 9th ACM Int'l Conf. on Distributed Event-Based Systems*, 2015, pp. 336–339.

[6] R. Pathak and V. Vaidehi, "An efficient rule balancing for scalable complex event processing," in *Proc. 2015 IEEE 28th Canadian Conf. on Electrical and Computer Engineering*, 2015, pp. 190–195.

[7] N. Zygouras, N. Zacheilas, V. Kalogeraki, D. Kinane, and D. Gunopulos, "Insights on a scalable and dynamic traffic management system," in *Proc. 18th Int'l Conf. on Extending Database Technology*, 2015, pp. 653–664.

[8] Y. Mei and S. Madden, "ZStream: A cost-based query processor for adaptively detecting composite events," in *Proc. 2009 ACM SIGMOD Int'l Conf. on Management of Data*, 2009, pp. 193–206.

[9] M. Proctor, "Drools: A rule engine for complex event processing," in *Proc. 4th Int'l Conf. on Applications of Graph Transformations with Industrial Relevance*, 2012, pp. 2–2.

[10] S. Burer and A. N. Letchford, "Non-convex mixed-integer nonlinear programming: A survey," *Surveys in Operations Research and Management Science*, vol. 17, no. 2, pp. 97–106, 2012.