Elastic Complex Event Processing exploiting Prediction

Nikos Zacheilas, Vana Kalogeraki
Department of Informatics
Athens University of Economics and Business
Athens, Greece
{zacheilas, vana}@aueb.gr

Nikolas Zygouras, Nikolaos Panagiotou, Dimitrios Gunopulos
Department of Informatics
University of Athens
Athens, Greece
{nzygouras, npanagiotou, dg}@di.uoa.gr

***Abstract*—Supporting real-time, cost-effective execution of Complex Event processing applications in the cloud has been an important goal for many scientists in recent years. Distributed Stream Processing Systems (DSPS) have been widely adopted by major computing companies as a powerful approach for large-scale Complex Event processing (CEP). However, determining the appropriate degree of parallelism of the DSPS' components can be particularly challenging as the volume of data streams is becoming increasingly large, the rule set is becoming continuously complex, and the system must be able to handle such large data stream volumes in real-time, taking into consideration changes in the burstiness levels and data characteristics. In this paper we describe our solution to building elastic complex event processing systems on top of our distributed CEP system which combines two commonly used frameworks, Storm and Esper, in order to provide both ease of usage and scalability. Our approach makes the following contributions: (i) we provide a mechanism for predicting the load and latency of the Esper engines in upcoming time windows, and (ii) we propose a novel algorithm for automatically adjusting the number of engines to use in the upcoming windows, taking into account the cost and the performance gains of possible changes. Our detailed experimental evaluation with a real traffic monitoring application that analyzes bus traces from the city of Dublin indicates the benefits in the working of our approach. Our proposal outperforms the current state of the art technique in regards to the amount of tuples that it can process by four orders of magnitude.**

I. Introduction

Nowadays we observe a large increase in the amount of data that needs to be analyzed and processed in real-time in a variety of application domains, ranging from traffic monitoring [1] to healthcare infrastructures [2] to financial processing [3]. Complex Event Processing systems (CEP) have emerged as a valid solution for analyzing high volume of information and detecting events of interest. In current CEP systems, users define rules that process *primitive* events received from a monitored environment and detect *composite* phenomena by combining primitive or other composite events using a set of event composition operators. An example of a most commonly used CEP is Esper[1], a system that has also been adopted by companies such as Oracle and Huawei.

Despite the plethora of different CEP engines that can be used, achieving scalability while maintaining high throughput has been considered a major challenge. For this reason,

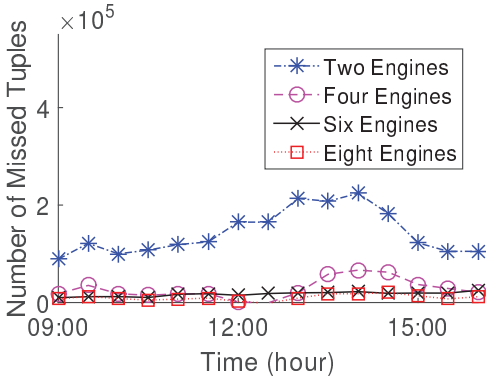

Fig. 1. Number of missed/unprocessed tuples, using different number of Esper engines.

Distributed Stream Processing Systems (DSPS) like Storm[2], Infosphere Streams[3] and Spark[4] have been applied for complex event detection. These systems offer scalable and low latency data processing by distributing the computation between multiple concurrently running components. However, in a stream processing system automatically scaling running applications can be particularly challenging as it is hard to predict future load dynamics, estimate the impact of the upcoming load in the system's performance or the appropriate number of resources to use [4]. Furthermore, most of the DSPS lack the expressiveness and ease of use of CEP systems like Esper, which provides an SQL-like language, EPL (Event Processing Language) for expressing the rules.

An example of the needs of automatically scaling complex event processing systems comes from the traffic monitoring domain. Today's technological advances are revolutionizing the way urban traffic information is collected and utilized to identify events of interest which meet the needs of businesses, residents and commuters. We have developed the INSIGHT system [5], whose goal is to identify emergency events in Dublin city, monitoring voluminous and heterogeneous data streams. In Dublin, the main mode of public transport is by bus; where buses are equipped with Automatic Vehicle Location (AVL) systems to track their fleet[5]. The real-time GPS traces from buses, described in Table I, are correlated with

[1] http://www.espertech.com/esper/

[2] https://storm.apache.org/

[3] www.ibm.com/software/products/us/en/infosphere-streams

[4] https://spark.apache.org

[5] http://dublinked.com/datastore/datasets/dataset-304.php

978-1-4799-9926-2/15/$31.00 ©2015 IEEE

Property	Value
Number of buses	911
Size of data	160 MB per day
Number of lines	67
Data frequency	3 tuples/min per bus
Time interval	6am till 3am

TABLE I. DATASET PROPERTIES

data collected from social networks (e.g. tweets referring to traffic issues) to more accurately identify the source and nature of the event. Figure 1 illustrates that several hundreds of tuples fail to be processed within their specified time window by a CEP system that processes traffic data and detects abnormal traffic conditions when different number of Esper engines are used during the execution (we provide more details about the experiment and the rule we used in Section IV). The data volume as well as the complexity of the rule varies during the course of the day. As the figure illustrates, increasing the amount of Esper engines can lead to a significant decrease in the amount of unprocessed tuples. If our system is capable of automatically scaling up during these hours (e.g. use more Esper engines in parallel) we can keep its performance steady by processing all the incoming data and avoid situations such as the one observed in Figure 1 where only two Esper engines are inadequate to process the incoming tuples.

To deal with this problem, one of the most commonly applied techniques is elasticity [4]. With this approach, when an application exhibits load spikes, some nodes in the application graph are replicated in order to balance the load across multiple nodes, and thus, improve the overall system throughput. This procedure requires more processing resources for the bursty application but effectively reduces the impact of the load spike. Elasticity in DSPSs has mainly been studied in [6] and [7]. These techniques examine the problem reactively trying to adjust the replication degree when a burst occurs. However, a pro-active technique which detects the incoming load and adjusts the components' parallelism using only minimal resources could augment the system's performance. Another commonly applied technique for this problem is load shedding [8]; however this approach randomly discards data from an application when increased load occurs, causing important information loss.

In this paper we describe our approach towards building elastic complex event processing systems on top of our distributed CEP system [5] which combines two widely used frameworks, Storm and Esper. Our approach aims to detect sudden changes in the input rate or latency increases, and determine an appropriate system policy with respect to the utilized resources for handling these sudden changes. The core of the method is the application of machine learning algorithms used for forecasting, in order to exploit periodic information observed in the data streams. Using the forecasting results, a graph of system states is created representing the short term future. Each edge of the graph represents an action to be taken and has a weight according to the estimated transition cost. This cost depicts possible loss of information due to lack of processing units or waste due to unnecessary resources. We exploit this graph in order to detect the sequence of actions that lead to the smallest overall cost.

In this work we focus on the problem of dynamically determining the appropriate number of Esper engines to handle load spikes, while at the same time minimizing the required resources. Although we implement and test our approach in a realized system that employs Storm and Esper, we emphasize that the techniques we develop and the problems we address are general and can be applied to a wide range of distributed CEP systems with similar characteristics. Our proposal consists of the following contributions:

- We apply Gaussian Processes for the estimation of the upcoming load and we compute the expected latency of the Esper engines.
- We invoke our estimation model for multiple time windows ahead of the current state, examining the behavior of the system when different number of Esper engines are used. We model the possible transitions (i.e. increasing/decreasing the number of engines) via a state transition graph. We detect the appropriate transitions that will keep the performance steady and at the same time will not over-utilize the system's resources.
- Finally, we evaluate our approach using a traffic monitoring application that detects traffic events in the city of Dublin exploiting the dataset described in Table I. Our experimental results illustrate that our approach is practical, exhibits good performance, and effectively adjusts in real-time the number of Esper engines reducing the amount of unprocessed tuples by four orders of magnitude, compared to the current state of the art technique [9].

II. LAMBDA ARCHITECTURE

In this section we give an overview of the basic components of our CEP framework described with more details in [5]. We also describe how we combined them to support a scalable and easy to use CEP for supporting a high volume of incoming data. Our framework can be seen as an instance of the Lambda Architecture[6] where incoming data are received via a DSPS and forwarded to concurrently running CEP engines.

A. Storm

Storm is one of the most commonly used DSPS, utilized by companies such as Twitter and Spotify. It has been employed for a wide range of applications, processing high volume data with low response times [10]. Processing this huge amount of information in real-time is achieved by distributing the workload across multiple machines. An application in Storm is represented by a *topology*, which is a graph where nodes are components that encapsulate the user-defined processing and edges model data flows among components. Components of a topology can be either *spouts* or *bolts*. Spouts represent the input sources which supply the topology with data, while bolts encapsulate the processing logic.

In Figure 2 we illustrate the architecture of the Storm framework in our setting. We enhanced Storm by enabling the user to submit topologies via an XML file. In the XML, the user decides the components that will be used and the communication patterns between them. From an architectural perspective, Storm follows a Master-slave architecture. The Master

[6]lambda-architecture.net

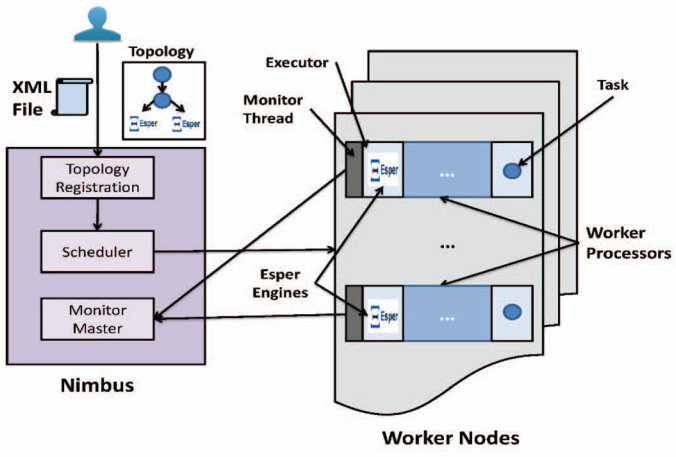

Fig. 2. Our System Architecture.

node, called Nimbus in the Storm terminology, is responsible for receiving topologies and scheduling their execution in the available worker nodes. Each node is equipped with multiple *worker processors* that receive and execute the components' implementation. Usually the number of worker processors is equal to the available CPU cores in the corresponding node.

As you can observe in Figure 2, each worker processor can execute multiple concurrently running *executors*. Executors are the threads that run the user defined code for a component (i.e. either bolt or spout). All the executors running in a processor belong to the same topology. The actual processing is performed by the components' *tasks*. Tasks are Java objects containing the user-defined code for the components and are assigned to executors for the actual execution in the Storm cluster. Ideally there should be one executor for each task. Users in our framework can specify the number of executors, tasks and worker processors via the XML file.

B. Esper

Esper is a Complex Event Processing (CEP) system, applied to streaming data, that triggers actions when incoming data satisfy some predefined rules. The core of the Esper system is the Esper engine which consists of a set of standing queries (or rules). Esper differentiates from most Big Data technologies (e.g. Hadoop[7]) in the fact that it does not store incoming data, but rather keeps them in an in-memory buffer, decreasing significantly the rules' execution times. Esper stores rules in the Esper engine and when new data arrive checks whether or not these rules are fired. This procedure is continuous, as incoming data are processed serially and the Esper engine responds in real time if any of the stored events meets the constraints.

Triggered events can be pushed further into the Esper engine feeding other rules or sent to their *listeners*. Listeners define the actions to be taken when the rule is activated. Users can create queries and add them into the Esper engine. Rules are written in Event Processing Language (EPL) and their syntax is similar to *SQL* queries. Each EPL query defines a sliding or batch window of the incoming stream that it monitors. We exploit the expressiveness and ease of usage of EPL by creating a special type of bolts in our Storm topologies that contain Esper engines (as you can see in Figure 2). Each bolt's task runs its dedicated Esper engine. Users define the rules that will be assigned to their Esper engines via the XML file they provide for their topology.

[7]https://hadoop.apache.org/

III. Methodology

The goal of this work is to dynamically adjust the number of Esper engines used by our system for processing incoming events without information loss (i.e. unprocessed tuples) and using the minimum required computing resources. Our approach (described in detail in the next sections) consists of the following steps:

1) We monitor in real-time the amount of incoming tuples and the tuples' end-to-end execution times (i.e. latency), and estimate their expected values in upcoming time windows.
2) We exploit these estimations by constructing a state transition graph that depicts the impact of possible actions (i.e. adding/removing Esper engines) to the system's performance (i.e. amount of missed tuples) and the required computing resources (e.g. VMs that will run the Esper engines).
3) We choose the appropriate transition actions in the examining horizon by finding the shortest path in our state transition graph. Our goal is to minimize a cost function that considers both the required computing resources and the system's performance.
4) Finally, we provide a technique for keeping and retrieving the engines' states (i.e. in-memory data) when their number changes. We apply this mechanism to avoid losing data during the state transitions.

A. Our System's Parameters

The Storm topology used in our system consists of the input sources (e.g. bus data stream) which emit incoming data to multiple concurrently running Esper engines that execute user-defined Esper rules. In order to be able to dynamically decide the number of engines to use, we have to estimate the expected number of input tuples and the tuples' latency. So we gather data while the topology runs in the cluster by adding special monitor threads in the Storm architecture, similar to [11]. We define the following parameters for our Storm topology:

- Δt: the monitor threads' sampling period. Monitor threads gather performance metrics (e.g. input tuples, latency) during this period and emit them to Nimbus when the time has elapsed. Nimbus aggregates this information to gather the complete performance metrics about the components.
- ES: the number of active Esper engines. Our aim is to adjust this metric dynamically based on the upcoming system's load and tuples' latency.
- ES_{max}: the maximum number of Esper engines that can be used by the topology for processing the incoming tuples. We use one Esper engine per cluster node, so this metric is limited by the available cluster computing resources (e.g. number of VMs).

- MC: represents the cost of missing tuples during the processing. Tuples can be missed if the current processing resources (i.e. Esper engines) cannot handle the incoming load. This a user-defined metric as we enable the user to decide the relative importance of missed tuples in the topology. For example if the users' rules examine emergency situations (e.g. floods) loosing tuples can be critical so this cost should be high. On the other hand, if the application monitors traffic conditions in a city, lost tuples are not catastrophic so the missed tuples cost could be low.
- EC: the monetary cost of the computing resources (e.g. VMs) used by our topology.
- RC: the cost of rebalancing the topology (i.e. changing the number of engines). This metric depicts the time needed by our system to appropriately transfer the states of the engines (i.e. engines' in-memory data) before it continues processing incoming events.

In each sampling period Δt we gather the following performance metrics:

- Lat: the end-to-end tuple's latency. This metric is defined as the end-to-end time needed to process a single tuple, this starts from the time the tuple is transfered from the input source to the appropriate Esper engine and the processing time required by the corresponding engine.
- In: the input tuples in the examining sample period. This metric depends on the rate with which the input sources emit tuples in the Esper engines.
- sel: the selectivity of the processing rules implemented on the Esper engines, depicting the ratio of output to input rate.
- Out: the amount of processed tuples. This metric depends on the observed latency, the selectivity, the number of Esper engines and the input tuples. It is computed via the following Formula:

$$Out = sel \times min\{\frac{\Delta t}{Lat} \times ES, In\} \quad (1)$$

- $Miss$: the amount of missed tuples. We consider as missed the input tuples that were not processed in the the sampling period. This metric is computed with the following Equation:

$$Miss = In \times sel - Out \quad (2)$$

B. Estimation Component

A key challenge in our proposal is estimating the upcoming load and expected latency of the Esper engines. It is of great importance to make accurate predictions for the future system's condition. These predictions determine the model's decision to add or remove computational resources from the system's configuration. There is a great variety of regression techniques that could be used in order to make these estimations including Support Vector Machines (SVM), Neural Networks, Random Forests and polynomial regression [12]. We decided to model our problem using Gaussian processes (GP) [13] because they can help us estimate the uncertainty concerning our predictions. In addition, as it is described in Section IV, Gaussian processes provided improved accuracy in comparison to the other methods. A shortcoming of the GP is the high computation complexity of the training phase, $O(n^3)$, due to the matrix inversion [13]. For this reason we re-train our model during fixed non-busy time periods (e.g. every night). Finally, the kernel functions are optimized exactly, maximizing over the model's hyperparameters. In this section, we describe the GP model and we illustrate its application in our setting.

In order to estimate the future number of missed tuples $Miss$ (Equation 2) we are interested in predicting accurately the future values of the latency (Lat) and the input tuples (In), to be used in Equations 1 and 2. We want to predict these values for specific future timestamps w_1, w_2 ..., w_H and system resources. In order to do these estimations, we use historical data in order to train two different GP models. The **first GP model** will have as target variable y the Lat values and as features will use a multidimensional vector that contains the *timestamp* of the measurement, the *time of day*, the *day of week* and the *number of engines*. While the **second GP model** has as target variable y the In values and as features the previous features except from the *number of engines* that is not correlated with the input rate. We use these features in order to capture correlations between the target variables and both the previously identified data and the historical data. The *timestamp* will help to improve the predictions for short time periods. On the other hand, the *time of the day* and the *day of week* will improve both the sort and the long term predictions, as they will correlate the requested data points with the corresponding historical data that were observed at previous days and hours. Finally we selected the *number of engines* as feature, in the Lat prediction, because we observed that the average latency to process a tuple differs when we use different number of Esper engines.

Gaussian Processes: Gaussian Processes were used in order to build the two prediction models described above, in order to predict the system's future condition. The Equations described in this Section were applied for the two different models but with different values for features x and target variables y for each model. The Gaussian process [13] is a non-linear non parametric model and is an extension of the multivariate Gaussian distribution for infinite collection of real-valued variables. A GP is described from its mean function $m(x) = E[f(x)]$ and its covariance function $K(x, x') = E[(f(x) - m(x))(f(x') - m(x'))]$. The prediction for a new data point x_* is given from the following Gaussian distribution:

$$x_* \sim \mathcal{N}(\bar{f}(x_*), var(x_*)) \quad (3)$$

Where $\bar{f}(x_*)$ is the prediction for the target variable of the GP model at the new data point x_* and $var(x_*)$ is the variance of the estimation. The mean $\bar{f}(x_*)$ is given from Equations 4 and 5. The GP model is build using the observed data $\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, where x_i is a d-dimensional vector, describing the set of features and y_i is the corresponding target variable. Also K is the covariance matrix and is described in more detail bellow.

$$\bar{f}(x_*) = \sum_{i=1}^{n} \alpha_i k(x_*, x_i) \quad (4)$$

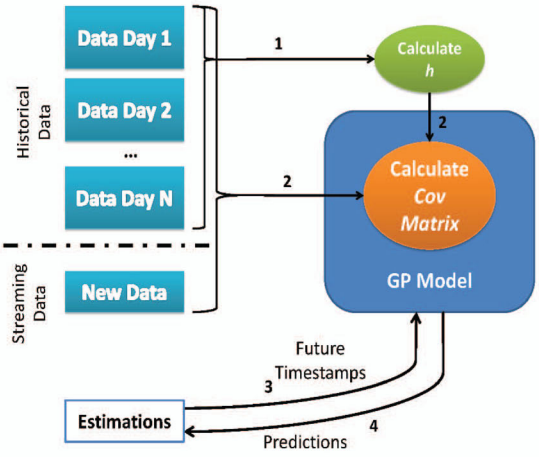

Fig. 3. Predictions Model: Step 1: calculate the hyperparameters, Step 2: compute the Covarience matrix, Step 3: request estimations for future timestamps, Step 4: return the estimations.

$$\alpha = (K + \sigma^2 I)^{-1} y \tag{5}$$

where σ is a hyperparameter of the model and the value of $\frac{1}{\sigma^2}$ represents the precision of the noise. The uncertainty concerning the estimations is provided from Equation 6. The n-dimensional vector $k(x_*)$ contains the distance of the new data point x_* with the observed data $x_1, \ldots, x_n$ and is defined via Equation 7. The scalar $k(x_*, x_*)$ is the distance from x_* with itself.

$$var(x_*) = k(x_*, x_*) - k^T(x_*)(K + \sigma^2 I)^{-1} k(x_*) \tag{6}$$

$$k(x_*) = (k(x_*, x_1), \ldots, k(x_*, x_n))^T \tag{7}$$

In order to create the covariance matrix K there are many covariance functions that could be used. These covariance functions characterize the correlations between different data points in the GP and they are **application dependent**. One widely used kernel function for multidimensional data is the *automatic relevance determination kernel* that is described below:

$$k(x, x') = \sigma_0^2 exp\Big\{ -\frac{1}{2} \sum_{d=1}^{D} \big(\frac{x_d - x'_d}{\lambda_d}\big) \Big\} \tag{8}$$

Different hyperparameters $h = \{\sigma_0, \lambda_1 \ldots \lambda_D\}$ result in a different GP. In general, greater values of σ_0 cause greater variance to the GP model. The advantage of the automatic relevance determination kernel function is that it provides different length scales λ_i for different dimensions. A higher value of λ_i decreases the importance of the respective dimension i making the corresponding feature irrelevant. It is very important to select the appropriate hyperpararemeters for each problem by maximizing the likelihood function.

Predicting System's Performance: In order to make predictions for each of the two models we followed a series of steps presented in Figure 3. These steps are described with more details bellow:

1) When the system is initialized we feed it once with the historical data. Then we calculate the automatic relevance determination kernel hyperparameters $h_1 = \{\sigma_0, \lambda_1, \lambda_2, \lambda_3, \lambda_4\}$ and $h_2 = \{\sigma_0, \lambda_1, \lambda_2, \lambda_3\}$ for the first and the second model respectively. The hyperparameters h_1 and h_2 are optimized by minimizing the negative log marginal likelihood w.r.t. the hyperparameters.
2) When the system runs online it updates each model's covariance matrix K, by adding the information provided from the currently extracted data regarding the system's performance (Lat and In as target variables and the required features for each model).
3) The system asks from the two GP models to make estimations for the target variables at specific future time points x_*, using the appropriate feature vectors per model.
4) In order to identify the distribution, for each GP model and at the query data points that refer to future timestamps, we evaluate Equation 3 at these time points x_*. This will result to two different Gaussian distributions one for each target variable (In and Lat). The mean of the distributions will be our estimations and the standard deviation will indicate the uncertainty of our estimations.

C. Elasticity Manager

The objective of our system is to be able to automatically adjust the utilized resources (i.e. number of active Esper engines) according to the future estimations. Therefore, we define a model that represents the current as well as the future states together with the associated costs and we use this for the decision making. Our model consists of the following components:

- A set of sliding windows $W = \{w_1, ..., w_H\}$ representing the future horizon H we are looking forward into the future. A window w_j represents a time period Δt_j.
- A finite set of states $S = \{s_1, s_2, .., s_{ES_{max}}\}$. We assume that at state s_i, i engines will be used. The initial state of the system is noted as $s_{init} \in S$. An artificial last state is defined as s_{last}.
- A finite set of actions $a_{ij} \in A$ where action a_{ij} moves the system from state s_i to state s_j.
- An acyclic directed graph of states $G = (V, E)$.
- A set of vertexes V. A state s_i and a window w_j defines a vertex v_{ij} representing the system state at a specific time. The initial vertex representing s_{init} is v_{init} while the artificial terminal vertex is v_{last}.
- A set of arcs E. An arc $e = (v_{ij}, v_{i'j'})$ where $j' = j + 1$ connects two states of the system over consecutive time windows w_j and $w_{j'}$. Such an arc represents the transition of the system from the state s_i to the state $s_{i'}$. This transition is mapped to the action $a_{ii'}$.
- A cost per arc e. Every arc $e \in E$ has an associated cost $C(v_{ij}, v_{i'j'})$ depending on its vertexes where v_{ij} is the origin vertex and $v_{i'j'}$ is the destination vertex.

1) State Graph and Shortest Path: As follows from the previous definition, the main component of our model is a Graph G that represents the states at different times through vertexes v_{ij} and the state transitions trough the arcs $e = (v_{ij}, v_{i'j'})$.

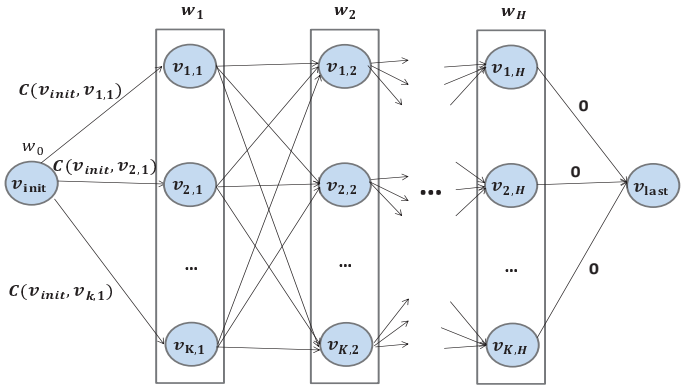

Fig. 4. States' Transitions Graph.

Using the graph of states, we could easily extract the shortest path π which directly translates to the optimal sequence of actions. This is done by using a graph search algorithm such as Dijkstra's Algorithm or A^* when the search space is too large. An example of our model is shown in Figure 4 where we have K possible states over a horizon of H windows. The system in the first window is at the initial state s_{init}. At each window we can move our system from v_{ij} to a different configuration $v_{i'j'}$ involving more or fewer engines but we should pay the appropriate cost for the shift defined by $C(v_{ij}, v_{i'j'})$. After running the shortest path algorithm we will have as a result the system states that keep the cost minimal. The cost estimation is crucial for the effectiveness of the detected path π.

2) Cost Estimation: For the cost estimation of an arc $e = (v_{ij}, v_{i'j'})$ the estimates about the future input tuples $\hat{In_{i'j'}}$ and maximum output tuples $\hat{Out_{i'j'}}$ are required at state $s_{i'}$ during the future window $w_{j'}$. $\hat{Out_{i'j'}}$ is defined from Equation 9, while the number of missed tuples $Miss_{i'j'}$ is calculated via Equation 10.

$$\hat{Out_{i'j'}} = sel \times min\{\frac{\Delta t_{j'}}{\hat{Lat_{i'j'}}} \times i', \hat{In_{i'j'}}\} \tag{9}$$

$$Miss_{i'j'} = sel \times \hat{In_{i'j'}} - \hat{Out_{i'j'}} \tag{10}$$

For every tuple miss there is a cost MC. In addition, every running engine has a cost EC proportion to the time Δt_j it is active. The assumption is that an engine is active for the whole duration Δt_j of a window w_j. The total number of engines used in a state s_i is i. Finally, every system state transition action involves a rebalancing cost RC defined in Equation 12 due to the latency caused by the migration of the Engines' states. As expected, when the state remains the same there is no rebalancing cost. Using the above parameters, the cost $C(v_{ij}, v_{i'j'})$ is defined from Equation 11.

$$C(v_{ij}, v_{i'j'}) = Miss_{i'j'} \times MC + i' \times EC \times \Delta t_j + RC(i, i') \tag{11}$$

$$RC(i, i') = \begin{cases} RC \text{ If } i \neq i' \\ 0 \text{ otherwise} \end{cases} \tag{12}$$

3) Path Rejection: Since the cost function defined in Equation 11 uses the expectation about the input tuples and observed latency, the cost itself is a random variable. According to the properties of the Gaussian Processes a prediction about the future is a random variable that follows also a Gaussian distribution under a specific mean μ and variance σ^2 as shown in Equation 13. As a result, the shortest path π comes with a standard deviation about the estimated cost of every arc. This standard deviation is useful for expressing our uncertainty about the prediction. If this uncertainty is high enough, we are restricted to use only the part of the path with the lowest uncertainty. Thus, even though the path gives as a sequence of actions for H sliding windows, only those with the least uncertainty will be considered. As shown in Algorithm 1, when the prediction error or the standard deviation of an arc e is high the current path is rejected and the optimization runs again.

$$C(v_{ij}, v_{i'j'}) \sim \mathcal{N}(\mu, \sigma^2) \tag{13}$$

Not using the total path horizon H will result in much more frequent elasticity optimizer calls. These calls are expected to increase the computation power needed for the elasticity model since a new graph is built from scratch. However, the rejection of uncertain paths enables the system to a-priory adapt to sudden load increases.

Data: Input Stream,w_0,s_{init},w_{init},v_{init},θ_1,θ_2
Result: Optimal policy π
$i \leftarrow s_{init}$;
$j \leftarrow w_{init}$;
$\pi \leftarrow null$;
$v \leftarrow v_{init}$;
while *!Stream.over()* **do**
  if $\pi.over()$ *or* $\pi = null$ **then**
    // Estimate future Input, Output
    $\hat{In} \leftarrow GP_{in}(j, H)$;
    $\hat{Out} \leftarrow GP_{out}(j, H)$;
    $G \leftarrow CreateGraph(\hat{In}, \hat{Out}, i, j, H)$;
    $\pi \leftarrow GraphSearch(G)$;
  end
  // Move to next window
  $j \leftarrow j + 1$
  // Get the next vertex using path π
  $v' \leftarrow \pi.nextVertex()$;
  // Move to new state
  $i \leftarrow State(v')$;
  // Check the cost uncertainty and prediction error
  if $Uncertainty(C(v, v')) > \theta_1$ *or* $Error(Stream_j, \hat{In_j}, \hat{Out_j}) > \theta_2$ **then**
    // Reject the path
    $\pi \leftarrow null$;
  end
  $v \leftarrow v'$;
end

Algorithm 1: Elasticity algorithm

D. Migrating Engines' State

Whenever the Elasticity Manager decides to add or remove engines, the system should properly adjust to the new condition without losing any information. The problem of keeping the state of processing components and migrating it appropriately to newly added, is a well-known problem in CEP and DSPS [14], [15]. We focus on the aspects of the problem when engines' data should migrate between the running engines and a newly added one. Each engine keeps multiple in-memory data that need to be migrated. It is of great importance to

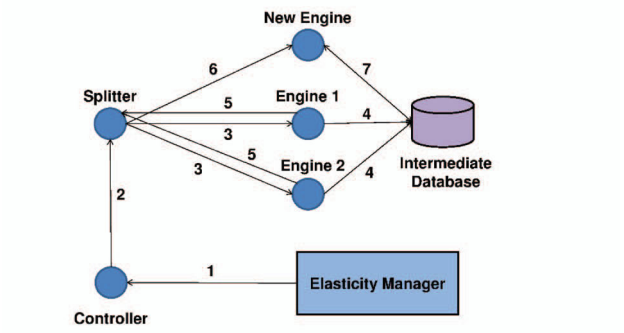

Fig. 5. Migrating Engines' State.

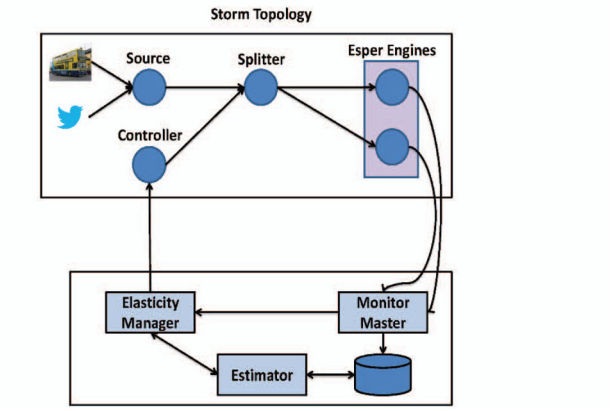

Fig. 6. System Implementation.

ensure that all the data needed for the new partition schema will be transmitted properly to the appropriate engines.

Inspired by the work in [15] we added two user-transparent components that help us guarantee the state of the Esper engines between transitions. The first component we added is the *Controller* spout which receives the transitions that must be applied in the upcoming time windows from the Elasticity Manager (*Step 1* in Figure 5). The second component is the *Splitter* component which is responsible to guarantee the system's proper and continuous operation. Whenever it receives a signal from the Controller (*Step 2*) for a new state it pauses the transmission of the streaming data to the Esper engines and starts migrating engines' state between the active engines. Initially it identifies the set of tuples that should be retransmitted. Afterwards it forwards a message to each Esper engine (*Step 3*) requesting to: (i) Delete the set of data that are not needed any more, (ii) Store these data to an intermediate database (e.g. in a distributed MongoDB database[8]) (*Step 4*), (iii) Inform the Splitter that the data were removed properly (*Step 5*). When the Splitter receives a message back from the Esper engines it informs the newly added engines (*Step 6*) in order to retrieve the data from the intermediate database (*Step 7*). When this procedure is over the streaming data transmission starts again.

E. Implementation

We have implemented our proposal in our distributed CEP system [5]. You can see the basic components of our approach and their interactions in Figure 6. As we mentioned in Section III-A, we have enhanced Storm with a monitor mechanism for monitoring the performance of the components (i.e. spouts/bolts). The Monitor Master is responsible to receive monitoring data from the Esper engines and to inform the Elasticity Manager about these newly received data. The latter checks if we have already decided the number of engines that will be used in the upcoming time periods. If we have not made such a decision, we estimate the performance of the topology in the upcoming time windows via the Estimator. This component utilizes the techniques described in Section III-B for estimating the topology's input rate and observed latency in the upcoming time windows.

Based on these estimations the Elasticity Manager creates the possible states' graph and invokes the algorithm described in Section III-C. The detected transitions are sent to the Controller component running in the topology that enables the change of the running engines in real-time (i.e. see Section III-D for more details). A special case occurs when the Elasticity Manager has already decided the number of engines to use in the current time period. In such a case, we examine if the observed cost is similar to the expected. If we observe a large deviation we re-execute our proposed elasticity algorithm.

[8]https://www.mongodb.org/

IV. Evaluation

We have performed an extensive experimental study of our approach on our local cluster consisting of 8 VMs. Each VM had attached two CPU processors and 3,096 MB RAM. All VMs were connected to the same LAN and their clocks were synchronized with the NTP protocol. We used Storm 0.8.2, Esper 5.1 and MongoDB 2.6.5. We used a separate node in our cluster where Nimbus executed to avoid overloading one of the VMs. We evaluated our approach in a real-time scenario focused on traffic monitoring in the city of Dublin.

Workload. The case of study is the real-time detection of traffic issues in the city of Dublin using streaming data that originate from public buses. We collected the data in the INSIGHT system (http://www.insight-ict.eu/) in the month of January 2013. Each bus indicates its position and the seconds that it is delayed from the original route schedule every 20sec. There are 911 distinct buses that start operating gradually at 06:00 and stop at 00:00. The elasticity of such a system is highly desirable in cases that traffic inspectors want to quickly reprocess historical data in a fast-forward perspective. In such cases the input rate increases orders of magnitude. The Esper engine received streaming data and the Splitter, according to the number of engines used, separates the city in corresponding number of areas and sends the streaming data to the appropriate Esper engine. Each Esper engine was responsible to identify abnormalities for each area. The Esper rule that we applied checked if more than 10 buses have continuously high delay values and whether at some time instance inside the examined window their distance was less than 10m.

Estimation Component. In the forecasting module we decided to use Gaussian Processes for the predictions (i.e. see Section III-B for more details). In this Section we provide sample results about the performance of different forecasting algorithms on predicting the future system's latency. The dataset comprised 2000 instances containing historical data. From

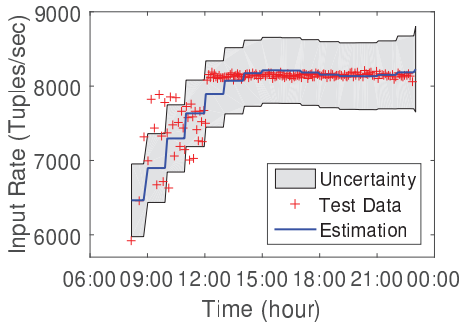

Fig. 7. Estimation of the input tuples.

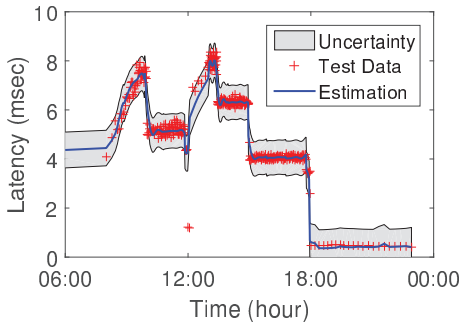

Fig. 8. Latency Estimation - Two engines.

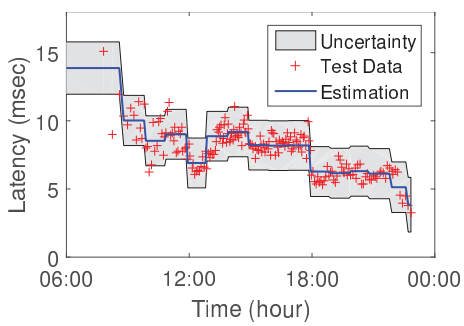

Fig. 9. Latency Estimation - Eight engines.

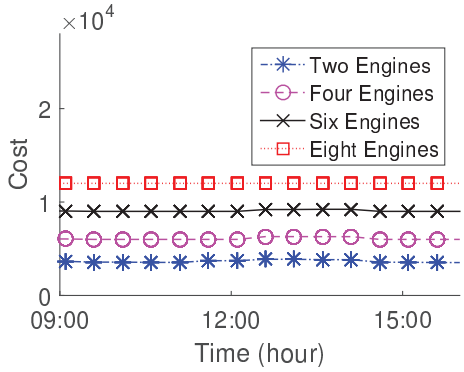

Fig. 10. $MC = 0.001$ - $EC = 1500$.

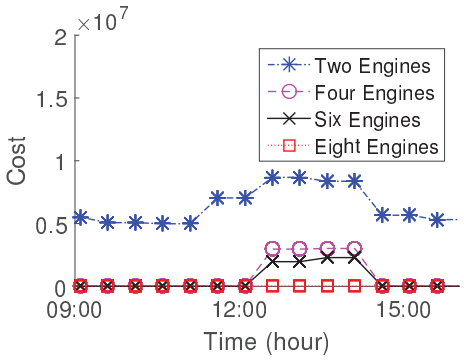

Fig. 11. $MC = 10$ - $EC = 1500$.

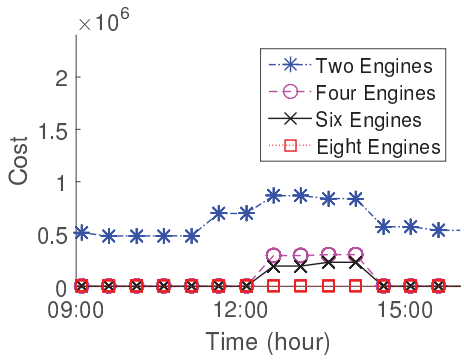

Fig. 12. $MC = 1$ - $EC = 0$.

these, the first 1400 were selected for training while the rest were used as a testing set. The results are reported on terms of Correlation Coefficient(CC), Mean Absolute Error(MAE), Root Mean Squared Error(RMSE) and Training Time(TT). The algorithms included a non-linear Support Vector Machine (SVM) using a Radial Basis Function (RBF) kernel, a Neural Network with automatically selected number of hidden layers and the Gaussian Process described in Section III-B.

The results are presented on Table II and suggest that the performance of the SVM and Neural Network are similar. However, since these are parametric methods their results highly depend on parameter tuning. The Gaussian Process scored the lowest MAE and RMSE and the highest Correlation Coefficient. A drawback on the other hand is that the training time of the Gaussian Process is highly increased in comparison to the other two algorithms. It is important to note that again the training time is depended on the selected parameters as well as on the training data.

An example of the predictive power of the forecasting component is given in Figures 7, 8 and 9. In these Figures, estimates about the future states of the system using a sample of the input streams are given. These estimations were used in all our experiments. The red points represent the actual state of the system in a specific future state. These points were not known during the time of the estimation. The Gaussian Process prediction is given by the blue line along with the estimation uncertainty notated by a gray surrounding area. It can be observed that the forecasting model could easily capture the periodic aspects of the datasets in all the three examples.

Focusing on Figure 7, it is important to note the red points that are highly deviating both from the prediction line (blue) as well as from the nearby actual (red) data points. These points are outliers that out of sudden do not follow the dataset pattern. This is an example of a case where the elasticity policy will be probably rejected due to high deviations between predictions and reality. On the other hand, in Figures 8 and 9 the conditions are much more stable.

Algorithm	CC	MAE	RMSE	TT
Gaussian Process	**0.94**	**0.58**	**0.86**	11.56
Neural Network	0.86	0.83	1.27	3.1
SVM	0.85	0.88	1.33	**3.0**

TABLE II. EVALUATION OF DIFFERENT FORECASTING ALGORITHMS.

Elasticity Manager. In this section we describe the performance of the Elasticity Manager, that was described in Section III-C. More specifically we examine the impact of the system's parameters MC and EC on the decision taken from the elasticity manager. In Figures 10, 11, 12 and 13 we present the cost of using 2, 4, 6 or 8 engines at a specific timestamp (being at a vertex $v_{i'j'}$ of the graph). In this experiment, we illustrate the impact of MC and EC parameters on the cost estimation. We assume that the RC cost is stable as we focus only on the impact of the MC and EC parameters. In general we observe that small values in MC result in smaller cost values and to smaller number of engines (Figure 10). On the other hand, when we select larger values of MC then the system assigns less cost to larger number of Esper engines(Figure 11). In addition when we select smaller values for the EC parameter then the system tends to use more engines (Figure 12). If we select large EC instead, the Elasticity Manager assigns smaller costs when less engines are

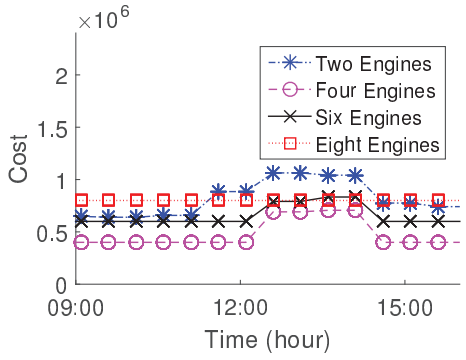

Fig. 13. $MC = 1$ - $EC = 100000$.

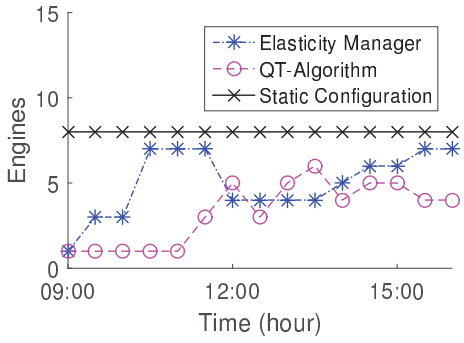

Fig. 14. Comparison of the number of engines used overtime for the different techniques.

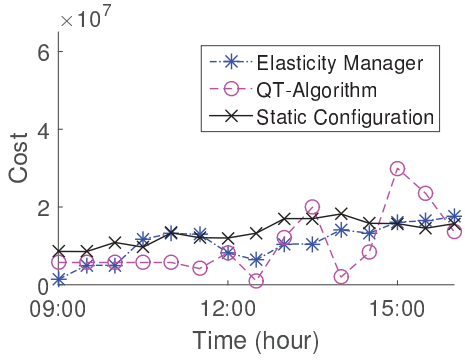

Fig. 15. Comparison of the observed cost overtime for the different techniques.

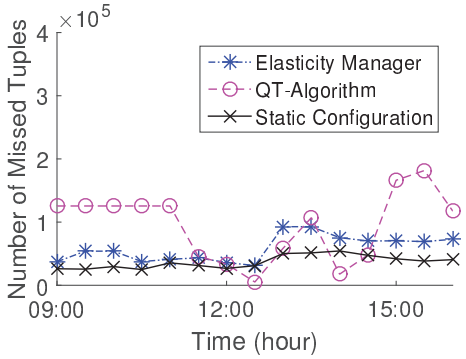

Fig. 16. Comparison of the missed tuples overtime for the different techniques.

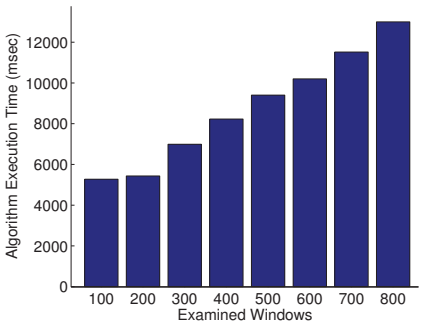

Fig. 17. Elasticity Manager's execution time for different number of examining windows.

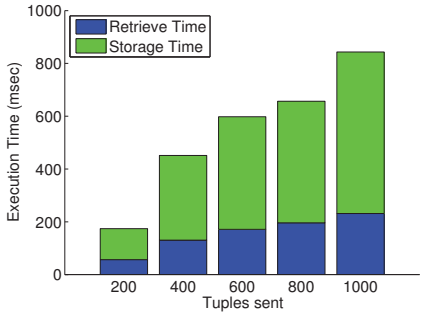

Fig. 18. State Migration Cost.

utilized (Figure 13).

Furthermore, we examined the performance of the Elasticity Manager in our local cluster. We compared our approach with the current state of the art method [9]. In this work, the authors propose the **QT-algorithm** which models the elasticity problem using queuing theory. They assume that the input rate follows a Poisson distribution while the processing time follows an exponential distribution. This assumption is not always true and as you can observe in our results it can affect significantly the observed cost and missed tuples metrics. Furthermore, we compared our approach with a static configuration that uses as much engines as possible during the course of execution. We performed this comparison in order to illustrate how close our proposal is in regards to the observed missed tuples with the approach that looses the least tuples but with maximum engines' cost. We display the different number of engines used during the topology's execution in Figure 14.

In Figure 15 you can see how our approach minimizes the observed cost (as it was defined in Section III-C2), and outperforms both the QT-algorithm and the static configuration approach. Furthermore, as you can observe in Figure 16 our approach is able to also keep the amount of missed tuples to similar levels with the static maximum resources configuration. The QT-algorithm is not able to capture appropriately the distribution of the processing time as it does not consider the complexity of the rules and the amount of utilized engines in the observed latency so as you can see in Figure 16, it fails to make the appropriate state transitions that will keep the amount of missed tuples at a low level. On the other hand, using always all the cluster resources is not beneficial as the user pays for more resources than needed in periods of low processing. Finally, we display the execution time of the Elasticity Manager's states' transition algorithm for different horizon sizes. As you can observe in Figure 17 the execution time depends on the number of windows we examine. Even for large window sizes the execution time of the algorithm is less than 12 seconds. Therefore, the overhead of the algorithm in regards to the execution time is negligible and will not affect the performance of the running applications.

Migrating Engines' State. In this experiment we illustrate the performance of our strategy that migrates an Esper engine's state. The data transmission among engines, discussed in Section III-D consists of two main steps the data storage step and the data retrieval step. In the first step the requested data are retrieved from the Esper engines and are stored in a database (i.e. a MongoDB in our case). The retrieval step consists of the $SELECT$-queries in the database from the Esper engines in order to acquire the requisite data. The times needed to store and retrieve data are presented in Figure 18. It is observed that the time increases for both storage and retrieval as the number of transmitted tuples increases. Also the time needed to store the data is always greater than the time needed to retrieve them.

V. Related Work

The authors in [16] focus their work on applying elastic Data Stream processing on top of the "FUGU" engine. They

formulate the problem as an online reinforcement learning algorithm where parameters about the state transition probabilities and rewards should be learned. They compare their approach against two baseline models involving "local" and "global" thresholds described in [17] and [18]. They suggest improved adaptivity on sudden bursts that may occur but also state that the method depends on learning parameters and tuning. A key difference with our work is that we take into account the monetary cost of the computing resources.

A similar reinforcement learning approach but using Markov Decision Processes instead, is described in [19]. In this case, the system has a set of states each of them representing a cluster size. Using the MDP model they periodically make a decision according to the current system measurements. In order to solve the MDP they use indirect methods where the optimal policy is found in a greedy manner using the Q-Learning algorithm. The above learning algorithms training time as well as their performance depend highly on the parameter settings. However, on our case the most influential component (i.e. Gaussian Processes) is completely non-parametric and automatically adapts to unseen streams.

An approach focused on cost-effective resource allocation is presented on [20]. The authors are interested on Publisher/-Subscriber services involving high volume streams. For that reason, they create a cost model based on Infrastructure as a Service (IaaS) framework perspective such as Amazon Web Services. They prove that the optimal resource allocation is an NP-Hard problem and they provide a sub-optimal solution by translating the resource allocation to the bin packing problem. This sub optimal solution is practical for real-time applications of very high volume streams. Similar to our case the authors also formally define the monetary cost of the resources. We differentiate from this approach in the fact that our optimization model uses predictions about future conditions.

A different perspective is given by [21], [22] where the cost of a system transition is highly considered. The authors try to adapt a cloud system configuration in cases of unexpected spikes. However, they focus especially on the latency caused by a system that is rebalancing. They estimate the expected latency caused during a shift and then decide when to move to a new configuration. This approach is orthogonal to ours since their estimations about the rebalancing latency could be merged with our forecasts about the future conditions.

VI. Conclusions

In this paper we presented a novel approach for automatically scaling complex event processing systems. We applied Gaussian processes for estimating the upcoming load and performance of the system. We utilized these estimations for automatically adjusting the CEP engines that will be used for multiple upcoming time windows. Our goal was to meet both the performance goals (i.e. missed tuples) and at the same time do not over-utilize the system resources. Finally, our experimental results in our local cluster indicate a clear improvement in the system's performance when our proposed approach is applied. For future work, our plan is to examine the performance of our approach using datasets with more uncertain behavior and different cost policies (e.g. Amazon's EC2).

VII. Acknowledgments

This research has been financed by the European Union through the ERC IDEAS NGHCS project.

References

[1] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran, "IBM Infosphere Streams for Scalable, Real-time, Intelligent Transportation Services," *SIGMOD*, 2010.

[2] M. Liu, M. Ray, D. Zhang, E. A. Rundensteiner, D. J. Dougherty, C. Gupta, S. Wang, and I. Ari, "Realtime Healthcare Services Via Nested Complex Event Processing Technology," *EDBT*, 2012.

[3] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White, "Towards Expressive Publish/Subscribe Systems," *EDBT*, 2006.

[4] S. Schneider, M. Hirzel, and B. Gedik, "Tutorial: Stream Processing Optimizations," *DEBS*, pp. 249–258, Arlington, TX, USA, June 2013.

[5] N. Zygouras, N. Zacheilas, V. Kalogeraki, D. Kinane, and D. Gunopulos, "Insights on a Scalable and Dynamic Traffic Management System," *EDBT*, pp. 653–664, Brussels, Belgium, March 2015.

[6] S. Schneider, H. Andrade, B. Gedik, A. Biem, and K.-L. Wu, "Elastic Scaling of Data Parallel Operators in Stream Processing," *IPDPS*, 2009.

[7] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic scaling for data stream processing," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 25, no. 6, pp. 1447–1463, 2014.

[8] N. Tatbul, U. Çetintemel, and S. Zdonik, "Staying fit: Efficient load shedding techniques for distributed stream processing," in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 159–170.

[9] R. Mayer, B. Koldehofe, and K. Rothermel, "Meeting predictable buffer limits in the parallel execution of event processing operators," in *Big Data (Big Data), 2014 IEEE International Conference on*. IEEE, 2014, pp. 402–411.

[10] R. McCreadie, C. Macdonald, I. Ounis, M. Osborne, and S. Petrovic, "Scalable Distributed Event Detection for Twitter," *BigData*, 2013.

[11] L. Aniello, R. Baldoni, and L. Querzoni, "Adaptive Online Scheduling in Storm," *DEBS*, pp. 249–258, Arlington, TX, USA, June 2013.

[12] C. M. Bishop *et al.*, *Pattern recognition and machine learning*. Springer New York, 2006, vol. 4, no. 4.

[13] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2005.

[14] B. Gedik, "Partitioning functions for stateful data parallelism in stream processing," *VLDB*, pp. 517–539, 2014.

[15] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu, "Auto-Parallelizing Stateful Distributed Streaming Applications," *PACT*, 2012.

[16] T. Heinze, V. Pappalardo, Z. Jerzak, and C. Fetzer, "Auto-scaling techniques for elastic data stream processing," in *ICDEW*, Chicago, IL, USA, March 2014, pp. 296–302.

[17] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *SIGMOD*, 2013.

[18] V. Gulisano, R. Jimenez-Peris, M. Patino-Martinez, C. Soriente, and P. Valduriez, "Streamcloud: An elastic and scalable data streaming system," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 23, no. 12, pp. 2351–2365, 2012.

[19] A. Naskos, E. Stachtiari, A. Gounaris, P. Katsaros, D. Tsoumakos, I. Konstantinou, and S. Sioutas, "Cloud elasticity using probabilistic model checking," *arXiv preprint arXiv:1405.4699*, 2014.

[20] V. Setty, R. Vitenberg, G. Kreitz, G. Urdaneta, and M. v. Steen, "Cost-effective resource allocation for deploying pub/sub on cloud," in *ICDCS*, Madrid, Spain, June 30 - July 3 2014, pp. 555–566.

[21] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *DEBS*, Mumbai, India, May 2014, pp. 13–22.

[22] C. Lei, E. A. Rundensteiner, and J. D. Guttman, "Robust distributed stream processing," in *ICDE*, Brisbane, Australia, April 2013, pp. 817–828.