State-Aware Load Shedding from Input Event Streams in Complex Event Processing

Ahmad Slo, Sukanya Bhowmik, Kurt Rothermel

***Abstract*—In complex event processing (CEP), load shedding is performed to maintain a given latency bound during overload situations when there is a limitation on resources. However, shedding load implies degradation in the quality of results (QoR). Therefore, it is crucial to perform load shedding in a way that has the lowest impact on QoR. Researchers, in the CEP domain, propose to drop either events or partial matches (PMs) in overload cases. They assign utilities to events or PMs by considering either the importance of events or the importance of PMs but not both together. In this paper, we combine these approaches where we propose to assign a utility to an event by considering both the event importance and the importance of PMs. We propose two load shedding approaches for CEP systems. The first approach drops events from PMs, while the second approach drops events from windows. We adopt a probabilistic model that uses the type and position of an event in a window and the state of a PM to assign a utility to an event. We, also, propose an approach to predict a utility threshold that is used to drop the required amount of events to maintain a given latency bound. By extensive evaluations on two real-world datasets and several representative queries, we show that, in the majority of cases, our load shedding approach outperforms state-of-the-art load shedding approaches, w.r.t. QoR.**

***Index Terms*—Complex Event Processing, Stream Processing, Load Shedding, Approximate Computing, latency bound, QoS, QoR.**

I. Introduction

Complex event processing (CEP) systems are used in many applications to detect patterns in input event streams [1], [2], [3]. The criticality of detected patterns (also called complex events) depends on the application. For example, in fraud detection systems in banks, detected complex events might indicate that a fraudster tries to withdraw money from a victim's account. Naturally, the complex events in this application are critical. On the other hand, in applications like network monitoring, soccer analysis, and transportation [4], [5], [6], the detected complex events might be less critical. As a result, these applications might tolerate imprecise detection or loss of some complex events.

In CEP systems, input events are streamed continuously to CEP operators where the input events (or simply events) are partitioned into windows of events [1], [2]. Events within windows are processed by CEP operators to detect patterns (called pattern matching). A detected part of a pattern within a window is called a partial match (denoted by PM) where the partial match could become a complex event if the full pattern is matched. Within a window, there might exist several PMs at the same time where PMs represent an important part of the internal state of a CEP operator.

For most applications, it is important to detect complex events within a certain latency bound (LB) where the late detected complex events become useless [7], [8]. However, if the rate of input events exceeds the processing capacity of CEP operators, the input events queue up and the detection latency of complex events increases, possibly resulting in violation of the given latency bound. For CEP applications that tolerate imprecise detection of complex events and have limited processing resources, one way to keep the given latency bound is by using load shedding [5], [6], [9], [10]. Load shedding reduces the overload on a CEP operator by either dropping events from the operator's input event stream or by dropping a portion of the operator's internal state. This results in decreasing the number of queued events and in increasing the operator processing rate, hence maintaining the given latency bound.

Of course, load shedding may impact the quality of results (QoR) as it might falsely drop complex events (denoted by false negatives) or/and falsely detect complex events (denoted by false positives). Therefore, it is crucial to shed load with minimum adverse impact on QoR. In [5], [9], the authors propose two *black-box* load shedding approaches for CEP systems where their approaches drop input events that have the lowest utility. The approach in [5] uses event type and position within windows as features to probabilistically learn about the utility of events in windows. In [9], the event utility depends on the frequency of events in patterns and in the input event stream. In [6], [10], the authors propose two *white-box* approaches to perform load shedding in CEP where the focus is on dropping partial matches. However, the approach in [10] might also drop input events if the given latency bound might be violated. Both approaches depend on the following features to learn about the utility of PMs: the progress/state of the PM in the window and the number of remaining events in the window. These two features are used to predict the completion probability and the processing cost of the PMs and hence the PM utilities.

In the *black-box* approach, load shedding is performed in a finer granularity (event granularity), i.e., it drops individual events from windows, in comparison to *white-box* dropping approaches which mainly drop PMs, i.e, dropping in a coarser granularity. As a result, the white-box approaches might drop PMs that have relatively high utilities which adversely impacts QoR even if there exist events that may be dropped without

Ahmad Slo, Sukanya Bhowmik, and Kurt Rothermel are with the Department of Distributed Systems, Institute for Parallel and Distributed Systems, University of Stuttgart, Universitaetsstrasse 38, 70569 Stuttgart, Germany. E-mail: {ahmad.slo@ipvs.uni-stuttgart.de; sukanya.bhowmik@ipvs.uni-stuttgart.de; kurt.rothermel@ipvs.uni-stuttgart.de}

impacting QoR. On the other hand, the black-box approaches neither consider the importance nor the state of PMs. An event might have different utilities for individual PMs, depending on the importance and the state of PMs. As mentioned above, the work in [10] also drops events in overloaded cases where events with the lowest utilities might be dropped from all PMs. Events that belong to PMs with low utilities are considered to also have low utilities. However, low utility PMs might also contain highly important events. Hence, dropping these events might adversely impact QoR. Moreover, this approach is limited to skip-till-any-match pattern semantic [11].

In this paper, we extend our findings in [12] where we propose a new white-box load shedding strategy called hSPICE that combines the best of both black-box and white-box approaches. In particular, hSPICE is a white-box load shedding approach that drops events either from *windows* or from *PMs*– it sheds on the event-granularity– while considering the operator's internal state. In hSPICE, events have different utilities/importances for different PMs. Moreover, hSPICE supports all well-known CEP event operators and selection and consumption policies [13], [14], [15]. hSPICE predicts the utility of the events using a probabilistic model. The model uses the event type, the event position within a window, and the state of partial matches in a window to learn about the utility of events within windows. An important factor that influences the effectiveness of a load shedding approach is its overhead in performing the load shedding. A high load shedding overhead implies that a high percentage of the available processing power will be used to take the shedding decision. This results in reducing the available processing power to perform pattern matching, thus adversely impacting QoR. As we will show, hSPICE is a lightweight, efficient load shedding approach.

More specifically, our contributions in this paper are as follows:

- We propose a white-box load shedding approach for complex event processing called hSPICE. hSPICE performs load shedding at two granularity levels by dropping events either from windows or from PMs. hSPICE uses a probabilistic model to learn the utility of an event *for each PM* within a window. This event utility is then used to perform fine-grained event shedding from individual PMs. Additionally, hSPICE can perform event shedding at a coarser granularity, i.e., from windows, by using the utility of an event for all PMs within a window to learn the utility of the event within the window. As learning features, we use the type and position of the event within the window and the state of the PM.
- We provide an algorithm to estimate the number of events to drop to maintain the given latency bound. Additionally, we propose an approach that enables hSPICE to perform load shedding in a lightweight manner.
- We provide extensive evaluations on two real-world datasets and a representative set of CEP queries to prove the effectiveness of hSPICE and to show its performance, w.r.t. its adverse impact on QoR, in comparison to state-of-the-art load shedding approaches.

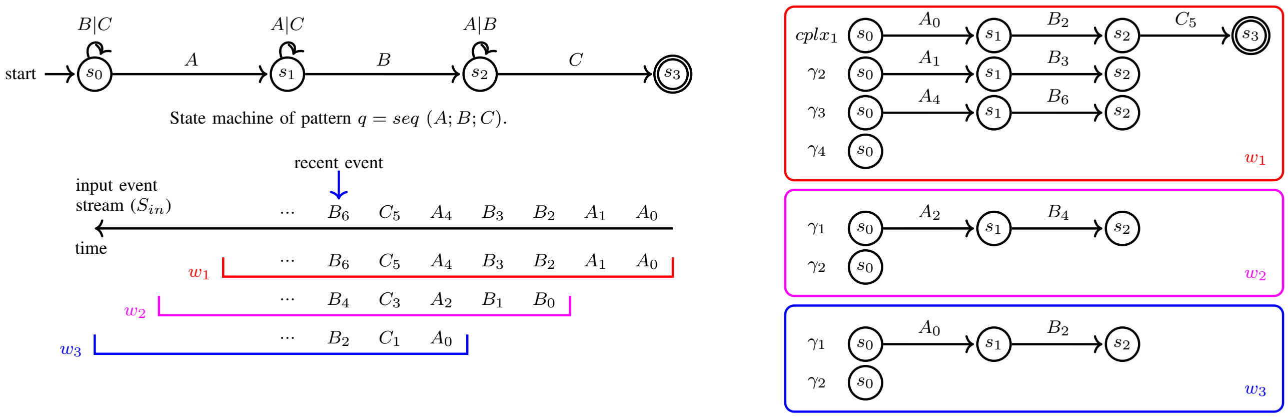
II. Preliminaries and Problem Statement

A. Complex Event Processing

A CEP system consists of a set of operators that are connected in the form of a directed acyclic graph (DAG). An operator in a CEP system correlates input events to detect patterns. The detected patterns are called complex events. An event in the input event stream (denoted by S_{in}) consists of meta-data and attribute-value pairs. The meta-data contains event type, sequence number and/or timestamp, while the attribute-value pairs represent the event data. For example, the type (denoted by T_e) of event e might represent a company name in a stock application, a player ID in a soccer application, or a bus ID in a transportation application. The event data might contain stock quotes, player positions, or bus locations in these applications. Events in the input event streams have global order, for example, by using the sequence number or the timestamp and a tie-breaker.

Our focus in this paper is on CEP systems consisting of a single operator, where the operator matches one or more patterns (i.e., multi-query). We define the set of patterns that the operator matches as $\mathbb{Q} = \{q_i : 1 \leq i \leq n\}$, where n is the number of patterns. Since patterns might have different importances, each pattern has a weight reflecting its importance. The patterns' weights are determined by a domain expert and they are defined as follows: $\mathbb{W}_{\mathbb{Q}} = \{w_{q_i} : 1 \leq i \leq n\}$, where w_{q_i} is the weight of pattern q_i. In CEP systems, the input event stream S_{in} is continuous and infinite, where the input event stream is partitioned into windows of events. Windows in CEP are opened depending on predicates such as time-based, count-based, or logical predicates. Moreover, the length of windows might be defined by time, event count, or logical conditions [2], [16]. The number of events in a window is defined as window size (denoted by ws). Each event in window w has a position where the position P_e of event e represents the number of events that precedes event e in window w. Windows might overlap which means that there may exist more than one open window at the same time. Hence, event e might belong to multiple windows, where it has different positions P_e within different windows. To clarify the system model, let us introduce the following example.

Example 1. In a stock application, an operator matches pattern q which correlates stock events from three companies. Pattern q is defined as follows: generate a complex event if a change in the stock quote of company A results in a change in the stock quote of company B, followed by a change in the stock quote of company C. We may write this pattern as a sequence operator [13]: $q = seq(A; B; C)$. Hence, the set of patterns that the operator matches is $Q = \{q\}$. In this example, the event type T_e might represent the company name, i.e., A, B, and C. Assume that a count-based predicate is used to open windows where a window is opened every two events, i.e., window slide size is two. Figure 1 depicts this example. Figure 1(a) shows that events in the input event stream (S_{in}) are ordered by the sequence number. Moreover, it shows that there are three open windows which overlap. Event A_4, for example, in S_{in} represents an instance of event type A. As an example to show how the same event may have different



B|C
A|C
A|B
start
s_0
A
s_1
B
s_2
C
s_3
State machine of pattern $q = seq\ (A; B; C)$.
recent event
input event stream (S_{in})
time
... B_6 C_5 A_4 B_3 B_2 A_1 A_0
w_1 ... B_6 C_5 A_4 B_3 B_2 A_1 A_0
w_2 ... B_4 C_3 A_2 B_1 B_0
w_3 ... B_2 C_1 A_0
$cplx_1$ s_0 A_0 s_1 B_2 s_2 C_5 s_3
γ_2 s_0 A_1 s_1 B_3 s_2
γ_3 s_0 A_4 s_1 B_6 s_2
γ_4 s_0
w_1
γ_1 s_0 A_2 s_1 B_4 s_2
γ_2 s_0
w_2
γ_1 s_0 A_0 s_1 B_2 s_2
γ_2 s_0
w_3

(a)

(b)

Fig. 1: Example 1.

positions within different windows, we see that the event A_4 from the input event stream belongs to all three windows, where it has the positions 4, 2, and 0 within windows w_1, w_2, and w_3, respectively.

Windows of events are first pushed to the input queue of a CEP operator. The operator continuously gets events from the input queue where, within every window to which an event belongs, the operator checks if the event matches the given pattern(s). We refer to this checking as processing the event within the window. As mentioned above, windows might overlap. However, events within each window are processed independently.

There exist several methods (a.k.a. computational models) to detect a pattern in CEP, e.g., finite state machine-based methods [1], [11], [14], [17], [18], tree-based methods [13], [19], [20], string-based methods [21], and Petri Nets-based methods [22]. To simplify the presentation and since finite state machine is the most commonly used computational model in CEP, in this work, we assume that a pattern in CEP is modeled as a finite state machine (cf. Figure 1(a)). Please note that our proposed load shedding approach is agnostic to the used computational model where we later show how our approach supports other computational models. The set of all possible states $\mathbb{S}_{q_i}$ of pattern $q_i \in \mathbb{Q}$ is defined as: $\mathbb{S}_{q_i} = \{s_k : j \leq k < j + m_i\}$, where m_i represents the number of all possible states of pattern q_i and j represents the sum of the number of all possible states of all patterns $q_l \in \mathbb{Q}$ where $l < i$, i.e., $j = \sum_{l=1}^{i-1} m_l$. In Example 1, pattern q has four states (i.e., $m_i = 4$) where $\mathbb{S}_q = \{s_0, s_1, s_2, s_3\}$ as shown in Figure 1(a). In the figure, s_0 represents the initial state of pattern q and s_3 represents its final state. We define the set of all possible states for all patterns as follows: $\mathbb{S}_{\mathbb{Q}} = \bigcup_{i=1}^{n} \mathbb{S}_{q_i}$. In Example 1, since there is only one pattern (i.e., $\mathbb{Q} = \{q\}$), $\mathbb{S}_{\mathbb{Q}} = \mathbb{S}_q = \{s_0, s_1, s_2, s_3\}$.

Whenever an operator starts to process events within a window, it starts an instance of the state machine of every pattern $q_i \in \mathbb{Q}$ at the initial state. During event processing within a window, an event is matched with the state machine instances of pattern $q_i \in \mathbb{Q}$. The event might cause the state machine instance(s) of pattern q_i to transit between different states of $\mathbb{S}_{q_i}$. Please recall that we have already defined a partial match. However, let us define it more formally. An instance of the state machine of pattern q_i is called a partial match (short PM), where the partial match completes and becomes a complex event if the state machine instance transits to the final state. Hence, processing an event within a window implies that the event is matched with PMs within the window. We define a partial match γ of pattern q_i as $\gamma \subset q_i$. Moreover, we refer to matching event e with PM $\gamma \in q_i$ as processing event e with PM γ, denoted by $e \otimes \gamma$. In Example 1, assume that the operator matches the events in windows chronologically [13] and the operator has already processed all available events in all open windows, i.e., the operator has processed the last event of type B (B_6 in the input event stream) in all windows. Figure 1(b) shows the result of pattern matching in all windows. In window w_1, the operator has detected one complex event ($cplx_1$) while there are still three open PMs in window w_1: γ_2, γ_3, and γ_4 . Similarly, there are two PMs in windows w_2 and w_3 each: γ_1 and γ_2.

Partial match $\gamma \subset q_i$ might be at any state of pattern q_i except the final state, where PM γ at the final state has already completed and become a complex event. Therefore, the set of all possible states ($\mathbb{S}_\gamma$) of PM γ is defined as follows: $\mathbb{S}_\gamma = \mathbb{S}_{q_i} \setminus \{\mathit{final\ states}\}$. Hence, the set of all possible states $\mathbb{S}_\Gamma$ of all PMs of all patterns is defined as follows: $\mathbb{S}_{\mathbb{\Gamma}} = \bigcup_{i=1}^{n} \mathbb{S}_{\gamma_i} : \gamma_i \subset q_i$. In example 1, for PM $\gamma \subset q$, $\mathbb{S}_\gamma = \{s_0, s_1, s_2\}$ and $\mathbb{S}_{\mathbb{\Gamma}} = \mathbb{S}_\gamma = \{s_0, s_1, s_2\}$, as there is only one pattern in this example. We refer to the current state of PM γ as S_γ. Additionally, we refer to PM γ at state s as γ_s. If processing event e with PM $\gamma \subset q_i$ at state s (i.e., $e \otimes \gamma_s$) causes γ to progress, i.e., e matches q_i and causes the state machine instance to transit, we refer to this as event e *contributes* to PM γ at state s, denoted by $e \in \gamma_s$. In Example 1, event B_0 in window w_2 has been processed with γ_1 at state s_0 (i.e., $B_0 \otimes \gamma_{1_{s_0}}$) but it did not cause γ_1 to progress. While in the same window w_2, event A_2 has been processed with γ_1 at state s_0 (i.e., $A_2 \otimes \gamma_{1_{s_0}}$) and it caused γ_1 to progress to state s_1. Hence, event A_2 contributes to PM γ_1 at state s_0, i.e., $A_2 \in \gamma_{1_{s_0}}$. In window w, at a certain window position P, there might exist one or more PMs belonging to the same or different patterns $q_i \in \mathbb{Q}$. We denote the set of PMs that are currently active at window position P by $\mathbb{\Gamma}_w^P$. Also, we denote the *total* number of PMs that are opened until the end of window w by $\mathbb{\Gamma}_w^T$. In Example 1 Figure1(b), the set of current PMs in windows w_1, w_2 and w_3 are as follows: $\mathbb{\Gamma}_{w_1}^6 = \{\gamma_2, \gamma_3, \gamma_4\}$, $\mathbb{\Gamma}_{w_2}^4 = \{\gamma_1, \gamma_2\}$, and $\mathbb{\Gamma}_{w_3}^2 = \{\gamma_1, \gamma_2\}$. Please note that in the negation operator [17], [18] if the negated event e' contributes to PM γ (i.e., $e' \in \gamma$), PM γ is abandoned. For ease of presentation, hereafter, we also refer to the abandoned PMs as completed PMs.

In CEP, there exist several event operators, e.g., sequence, negation, any, conjunction, disjunction, and Kleene closure operators [13], [14], [18], [20]. Moreover, there exist several selection and consumption policies, e.g., *first*, *last*, *each*, and *cumulative* selection policy and *zero* or *consumed* consumption policy [13], [14], [15]. Selection policies are used to determine exactly which event instances of the same event type should be used in detecting complex events, hence avoiding any ambiguity if event instances of the same event type occur many times in a window. While consumption policies determine whether an event that is already used in detecting a complex event is allowed to be reused in the detection of other complex events. We do not assume a specific event operator or a specific selection and consumption policy. In general, hSPICE supports the commonly used aforementioned event operators and selection and consumption policies.

B. Quality of Results

In this paper, we represent the quality of results (QoR) by the number of false positives and negatives. A false positive is a situation (a complex event) that should not be detected but has been falsely detected. While a false negative is a situation (a complex event) that should be detected but has not been detected.

There might exist several instances of each event type within a window, where the selection and consumption policy are used to exactly define which instance(s) of an event type must be used to detect complex events in the window. However, for many applications, it is sufficient to detect complex events regardless of the exact event instances that contribute to detect these complex events. Moreover, in many cases, the consecutive event instances of an event type represent only slight updates for the same event. Therefore, false positives and negatives can be defined in different ways depending on whether the application needs to match the exact event instances or not. In the following, we introduce two ways to define false positives and negatives, i.e, to define QoR.

Strict Quality of Results. In the strict quality of results, false positives and negatives are defined depending on the exact event instances. This type of QoR is important for applications in which the order of event instances or the causal relations between event instances are important. For example, in a security application, an employee opens a door with his/her ID card and there is a camera installed on the door. Hence, there are two event types: 1) event type ID indicates that the ID card opened the door, and 2) event type F represents a video frame. A CEP operator detects if the ID card that is used to open the door belongs to the same person (employee) who opened the door. Several persons might open the same door successively in a short time interval which means that there exist several instances of the ID event type ($T_e = ID$) and the frame event type ($T_e = F$). Dropping event instances of any of these two types might result in matching a wrong ID event with a wrong frame event. This might result in detecting that a different person opens the door (false positive) or detecting that a certain person has not opened the door (false negative). In another application, social networks for example, an analyst might be interested to detect which person has started a discussion on a certain topic. Let us assume that a person A has commented on a post. Then, a person B wrote a comment as a reaction to the comment of person A. After that, person A commented back. In this example, dropping event instances of the event types A and/or B might change the correct order of the comments. Hence, it might lead to incorrectly determine which person has started the discussion.

To define the strict QoR more precisely, in Example 1 (cf. Section II-A, Figure 1), let us consider window w_1 contains the following events (B_6, C_5, A_4, B_3 B_2, A_1, A_0). Each event type has one or more event instances in the window. For instance, the event type A has three event instances (i.e., A_0, A_1, and A_4) in window w_1. By processing window w_1, the operator detects a complex event $cplx_o$ from the events A_0, B_2, and C_5, i.e., $cplx_o = (A_0, B_2, C_5)$. Let us assume that due to load shedding, event B_2 is dropped from the window. In this case, the operator detects a new complex event $cplx_l$ from the events A_0, B_3, and C_5, , i.e., $cplx_l = (A_0, B_3, C_5)$. Since the new complex event $cplx_l$ is not detected from the same event instances as the complex event $cplx_o$, in the strict QoR, complex event $cplx_l$ is considered as a false positive. Moreover, as complex event $cplx_o$ is not detected in window w_1 due to load shedding, we count this case as a false negative. Hence, dropping event B_2 from window w_1 results in one false positive and one false negative.

Relaxed Quality of Results. In the relaxed quality of results, false positives and negatives are defined irrespective of the exact event instances, i.e., it is not important which instances of an event type contributed to detect a complex event. This type of QoR is useful for many applications, e.g., stock market, soccer, transportation, etc. For example, in a stock market application, stock events might come at a high frequency (e.g., every 1 minute), hence two consecutive stock events e and e' of a certain company (i.e., $T_e = T_e'$) might have a slight or even no difference in the stock quote (slight or no change in price). Therefore, to detect that a stock company A has influenced a stock company B in a certain time interval (window), it is enough to find a correlation between any event instance of stock company A and any event instance of stock company B in that time interval.

To clearly define relaxed QoR, in the above example, the newly detected complex event $cplx_l$ is considered equivalent to the complex event $cplx_o$. Hence, dropping event B_2 from window w_1 does not result in any false positive or negative in the case of relaxed QoR.

C. Problem Statement

A CEP operator might have limited resources where, in overload cases, it must perform load shedding by dropping a portion of the input events to avoid violating a given latency bound (LB). However, dropping events might degrade QoR, i.e., resulting in false positives and false negatives. Therefore, the load shedding must be performed in a way that has minimum adverse impact on QoR.

As we mentioned above, an operator might detect multiple patterns $\mathbb{Q}$ and each pattern has its weight (i.e., $\mathbb{W}_{\mathbb{Q}}$). For pattern $q_i \in \mathbb{Q}$, we define the number of false positives as FP_{q_i} and the number of false negatives as FN_{q_i}. The total number of false positives (denoted by $FP_{\mathbb{Q}}$) for all patterns is defined as the sum of the number of false positives for each pattern multiplied by the pattern's weight (cf. Equation 1). Similarly, the total number of false negatives (denoted by $FN_{\mathbb{Q}}$) for all patterns is defined as the sum of the number of false negatives for each pattern multiplied by the pattern's weight (cf. Equation 2).

$$FP_{\mathbb{Q}} = \sum_{q_i \in \mathbb{Q}} w_{q_i} * FP_{q_i} \tag{1}$$

$$FN_{\mathbb{Q}} = \sum_{q_i \in \mathbb{Q}} w_{q_i} * FN_{q_i} \tag{2}$$

As a result, the impact of load shedding on QoR is measured by the sum of the total number of false positives ($FP_{\mathbb{Q}}$) and the total number of false negatives ($FN_{\mathbb{Q}}$). The objective is to minimize the adverse impact on QoR, i.e., minimize ($FP_{\mathbb{Q}} + FN_{\mathbb{Q}}$), while dropping events such that the given latency bound LB is met. More formally, the objective is defined as follows.

$$\begin{aligned} minimize \quad & (FP_{\mathbb{Q}} + FN_{\mathbb{Q}}) \\ \text{s.t.} \quad & l_e \leq LB \quad \forall\, e \in S_{in} \end{aligned} \tag{3}$$

where l_e is the latency of event e that represents the sum of the queuing latency of event e and the time needed to process event e within all windows to which event e belongs.

III. Load Shedding in CEP

We extend a CEP operator with our proposed load shedding system (hSPICE) that in overload cases drops a portion of the input events to maintain the given latency bound (LB). In CEP, a load shedding system must perform the following three tasks: 1) deciding when input events must be dropped, 2) computing the time interval and the number of events that must be dropped in every time interval (denoted by drop interval) to maintain LB, and 3) dropping input events that have the lowest adverse impact on QoR. Tasks 1 and 2 have already been well studied in literature [5], [6]. Therefore, our focus in this paper is on task 3, i.e, deciding which events to drop. In the following, we shortly explain how tasks 1 and 2 might be performed. Figure 2 depicts a CEP operator extended with two components to enable load shedding: 1) overload detector and 2) load shedder (LS).

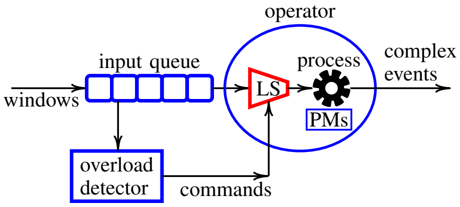


Fig. 2: The hSPICE Architecture.

The given latency bound (LB), the rate of incoming input events, and the operator throughput (maximum service rate) can be used as parameters to decide when to drop events. The overload detector periodically monitors these parameters. If the input event rate (R) is higher than the operator throughput (μ) for a long enough period, the given latency bound (LB) might be violated. To prevent violating LB, the overload detector requests the load shedder to drop a certain amount of input events. As a drop interval (λ), we might use the window size ws or a part of it as proposed in [5]. Our approach works with any drop interval. However, in this paper, to simplify the presentation, we consider that the drop interval equals the window size, i.e., $\lambda = ws$. The number of events that must be dropped in every window to maintain LB can be computed depending on the input event rate R and the operator throughput μ, where the overload detector computes the drop amount ρ per window (i.e., per drop interval) as follows: $\rho = (1 - \frac{\mu}{R}) * ws$. After that, the overload detector sends a command containing the drop interval λ and the number of events ρ to drop per λ to the load shedder. The load shedder drops ρ events per drop interval λ to maintain LB.

hSPICE

During overload, to maintain the given latency bound (LB), hSPICE drops input events that have the lowest adverse impact on QoR, i.e, on the number of false positives and negatives. To do that, hSPICE assigns utility values to the events where an event that has a high impact on QoR has a high utility and vice versa. hSPICE drops events either from windows (referred to as window granularity) or from PMs within windows (referred to as partial match granularity). Determining the utility of events on the PM granularity can be achieved more accurately since PM granularity is more fine-grained than window granularity. Of course, accurately predicting the event utilities might significantly reduce the adverse impact of load shedding on QoR. Another factor that influences the load shedding impact on QoR is the overhead of performing load shedding. A high load shedding overhead implies that more processing power is used by the load shedder, hence more events must be dropped which adversely impacts QoR. Performing load shedding on the window granularity imposes a lower overhead compared to performing load shedding on the PM granularity since the load shedding is performed on a coarser granularity. Therefore, there is a trade-off between accurately determining the event utilities and the load shedding overhead. In the next sections, for both window and PM granularities, we study how to predict the event utilities and analyze the imposed load shedding overhead on the operator.

On a high abstraction level, hSPICE works as follows. 1) As mentioned above, an event in a window is processed with

PMs within the window. Therefore, in a window, when using PM granularity, hSPICE assigns utility values to an event for each PM within the window individually, i.e., the event gets a certain utility value for each PM within the window. For the window granularity, on the other hand, hSPICE assigns only a single utility value to each event within the window, depending on the event utilities for PMs within the window. 2) hSPICE performs load shedding by dropping *events* either from *windows* (window granularity) or from *partial matches* within windows (PM granularity). Dropping an event from a window w means that hSPICE prevents processing the event with all current PMs ($\mathbb{\Gamma}_w^P$) within the window. While dropping an event from PM γ within a window means that hSPICE prevents processing the event with PM γ within the window.

hSPICE, primarily, performs two tasks: 1) model building and 2) load shedding. In the model building task, hSPICE predicts the event utilities and summarizes the event utilities to reduce the degradation in QoR in overload situations. In the load shedding task, hSPICE drops events to avoid violating the given latency bound. The model building task is not time-critical and can afford to be heavyweight. On the other hand, the load shedding task is time-critical and hence must be lightweight. In the next sections, for both window and PM granularities, we describe the above tasks in detail. First, we describe how the utility of an event is defined. Then, we explain the way hSPICE predicts the event utility using a probabilistic model. After that, we describe how hSPICE computes the number of events to drop to maintain the given latency bound. To perform load shedding efficiently, we explain how to predict a utility value that can be used as a threshold utility to drop the required number of events. Finally, we describe the functionality of the load shedder in hSPICE.

A. Partial Match Granularity

1) Event Utility: In a window, only some PMs might complete and become complex events. Hence, PMs in a window might have different importances, w.r.t. QoR. If a PM completes, it is an important PM for QoR. Otherwise, it has no impact on QoR. Moreover, as mentioned above, an event might be processed with one or more PMs within a window, where the event might contribute only to some of these PMs. An event that contributes to a PM might be an important event for the PM since dropping the event from the PM might hinder the PM completion and hence adversely impact QoR. On the other hand, an event that does not contribute to a PM is not important for the PM since dropping the event from the PM does not influence its completion. Therefore, for different PMs in a window, an event might have different importances. As a result, in a window, for event e and PM γ within the window, hSPICE assigns a utility value to event e (denoted by the utility of event e for PM γ) depending on the importance of PM γ in the window and on the importance of event e for γ. Higher is the importance of γ in the window and higher is the importance of event e for γ, higher is the utility of event e for γ.

The utility of event e for PM γ of pattern $q_i \in \mathbb{Q}$ within a window (denoted by $U_{e,\gamma}$) depends on three factors: 1) contribution probability—the probability that event e contributes to PM γ, i.e., $e \in \gamma$, 2) completion probability—the probability that PM γ completes, and 3) pattern weight w_{q_i} (given by a domain expert). Clearly, if event e has a high probability to contribute to PM γ, event e is an important event for PM γ. We consider the completion probability of a PM in computing the event utility as well since the PM is only useful if it completes. Therefore, if event e has a high probability to contribute to PM γ and γ has a high probability to complete, event e is an important event and should be assigned a high utility value. This is because dropping event e may hinder PM γ to complete and hence it may adversely impact QoR.

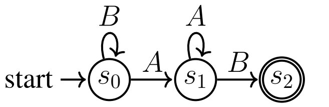
As a result, the utility $U_{e,\gamma}$ of event e for PM $\gamma \subset q_i$ within a window depends on the pattern weight w_{q_i} and the following probability: $P(e \in \gamma \cap \gamma \; completes)$, i.e., the probability that PM γ completes and event e contributes to PM γ. In window w, to predict $P(e \in \gamma \cap \gamma \; completes)$ and hence $U_{e,\gamma}$, hSPICE uses three features: 1) current state S_γ of PM γ, 2) event type T_e, and 3) position P_e of event e in window w. Therefore, the utility $U_{e,\gamma}$ of event e for PM γ of pattern q_i (i.e., $\gamma \subset q_i$) is defined as a function (called utility function) of these three features as shown in Equation 4:

$$U_{e,\gamma} = f(T_e, P_e, S_\gamma) = w_{q_i} * P(e \in \gamma \cap \gamma \; completes) \quad (4)$$

The current state S_γ of PM γ determines which event type(s) enables PM γ to progress, i.e., to transit to a new state(s). Therefore, those two features, i.e., current state S_γ of the PM and event type T_e are important features for computing $U_{e,\gamma}$. For instance, in Example 1, PM γ at state s_0 (i.e., γ_{s_0}), might transit to state s_1 only if event e of type $T_e = A$ is processed with PM γ (i.e., $e \otimes \gamma_{s_0}$).

The position P_e of event e in window w is an important feature to compute $U_{e,\gamma}$ as well since it determines the number of remaining events in the window. If there are still many events remaining in a window, the probability of a PM to complete might be higher than the case where there are only a few remaining events in the window. This is because, in case of many remaining events in a window, a PM has a chance to be processed with more events than in case of only a few remaining events in the window and hence the PM has a higher chance to progress. Moreover, the event position P_e represents the temporal distance between events within the same window. It determines which event instance(s) of the same event type has a higher probability to contribute to a PM in the window as shown in [5]. This is because there exists a correlation between events of certain types at certain positions within a window. A change in an event of a certain type influences the change of events of other types within a certain time interval, i.e., certain position(s) within the window. In Example 1, in window w, a change in the stock quote of company A, i.e., $T_e = A$, at a certain point of time t_1 (i.e., at a certain position in window), might cause a change in the stock quote of company B, i.e., $T_e = B$, within a certain time interval $]t_1, t_2]$, i.e., within certain position(s) in the window.

2) Predicting Event Utility: Having defined the utility $U_{e,\gamma}$ of event e for PM γ, now, we describe how hSPICE predicts the utility $U_{e,\gamma}$ within a window, i.e., $P(e \in \gamma \cap \gamma \; completes)$, hence predicting the value of utility function $f(T_e, P_e, S_\gamma)$ in Equation 4. For ease of presentation,



State machine for pattern $q = seq\ (A; B)$.

T_e/P_e	0	1	2	3	4
A	x	x	x	x	
B			x	x	x

TABLE I: Event distribution within windows.

A_0	$ob_e\langle 1-2, s_0, s_1, A_0\rangle : \frac{2}{6}$
A_2	$ob_e\langle 3-4, s_0, s_1, A_2\rangle : \frac{2}{4}$
A_3	$ob_e\langle 5-6, s_0, s_1, A_3\rangle : \frac{2}{2}$

B_3	$ob_e\langle 1, s_1, s_2, B_3\rangle : \frac{1}{4}$
B_4	$ob_e\langle 2-3, s_1, s_2, B_4\rangle : \frac{2}{5}$

$ob_\gamma\langle 1-3,\ completed\rangle$
$ob_\gamma\langle 4-6,\ not\ completed\rangle$

TABLE II: Contribution ob_e and completion ob_γ observations.

Fig. 3: Observations gathered from six PMs.

s_0					
T_e/P_e	0	1	2	3	4
A	33	0	25	0	0
B	0	0	0	0	0

s_1					
T_e/P_e	0	1	2	3	4
A	0	0	0	0	0
B	0	0	0	25	40

Fig. 4: Computing event utility $U_{e,\gamma}$ for a partial match.

we introduce a simple running example which is depicted in Figures 3 and 4.

Example 2. Let us assume that an operator matches a pattern $q = seq\ (A; B)$, where $\mathbb{S}_q = \{s_0, s_1, s_2\}$ and $\mathbb{S}_\gamma = \{s_0, s_1\}$, $\gamma \subset q$. The used window length is 5 events (i.e., $ws = 5$) and there are only two event types in the input event stream: A and B.

To predict the utility $U_{e,\gamma}$ of event e for PM γ of pattern q_i in window w, we first need to predict the *completion probability* of PM γ, i.e., find the probability that PM γ at state S_γ and at position P_e in window w will complete. Additionally, we need to predict the *contribution probability* of event e to PM γ, i.e., the probability that event e of type T_e at position P_e in window w contributes to PM γ ($e \in \gamma$). If the contribution and completion probabilities are high, then the event utility $U_{e,\gamma}$ is high. On the other hand, if the contribution and/or completion probabilities are low, then the event utility $U_{e,\gamma}$ is low. hSPICE uses statistics gathered over already processed windows to predict the completion and contribution probabilities, thus predicting the event utility for PMs. Next, we first show which statistics hSPICE gathers. Then, we explain the way the event utility $U_{e,\gamma}$ for PMs is predicted depending on those gathered statistics.

Statistic Gathering. To predict the contribution and completion probabilities (i.e., to predict $P(e \in \gamma \cap \gamma\ completes)$), thus predicting the value of utility function f, hSPICE gathers statistics on the progress of PMs within windows during event processing in an operator. To do that, hSPICE uses two types of *observations*: 1) contribution observation, denoted by ob_e, and 2) completion observation, denoted by ob_γ. In window w, for each event e within w, whenever event e is processed with PM γ at state $s = S_\gamma$ (i.e, $e \otimes \gamma_s$), the operator builds an observation of type contribution $ob_e\langle id, s, s', e\rangle$, where id is the id of PM γ. s' represents the state of PM γ after processing event e. If $s \neq s'$, event e has contributed to PM γ at state s, i.e., $e \in \gamma_s$. Additionally, in window w, if PM γ completes, the operator builds an observation of type completion $ob_\gamma\langle id, completed\rangle$, where again id is the id of PM γ. When window w closes (i.e., all its events are processed), all still open PMs in window w, i.e., Γ_w^P, (here P is the last position in w) are considered as *not completed* PMs.

Figure 3 shows an example of gathered observations on six PMs. Table I shows the distribution of event types in different positions within a window where a cell with x sign in the table means that the corresponding event type might be present at the corresponding position within a window. Please note that event types might not be present in all positions within a window. In the table, for example, the event type A never comes at position 4 in any window and event type B does not come at positions 0 and 1 in any window. Table II shows observations on event e of type T_e at position P_e in a window and PM γ at state s only if e contributes to γ (i.e., $e \in \gamma_s$). For example, in the table, event B_3 of type $T_e = B$ at position $P_e = 3$ within windows has never contributed to PM γ at state s_0. Therefore, there are no observations shown in the table on event B_3 with a PM at state s_0. Clearly, if event e is not present at a certain position within windows, event e can not contribute to any PM at this window position. For example, as shown in Table I, the event of type B never comes at position 1 within windows. Therefore, there are no observations on the event type B at position 1 within windows with a PM at any state. In Table II, next to each observation of type contribution ob_e, we show the number of PMs at state s to which an event *contributed* divided by the *total* number of PMs at state s with which an event is processed, i.e., $\frac{|\{e:e\in\gamma_s\}|}{|\{e:e\otimes\gamma_s\}|}$. For example, in the table, $ob_e\langle 3-4, s_0, s_1, A_2\rangle : \frac{2}{4}$ means that the event of type $T_e = A$ at position 2 within windows has been processed with four PMs at state s_0. However, it has contributed only to two PMs, in particular, it has contributed to PMs 3 and 4. The table also shows which PMs have completed. For example, in the table, PMs γ_1, γ_2, and γ_3 have completed while PMs γ_4, γ_5, and γ_6 have not completed.

After gathering statistics from η observations, hSPICE uses these observations to predict the utility $U_{e,\gamma}$ of event e for PM γ within window w, i.e., to predict the utility function f (cf. Equation 4).

Utility Prediction. hSPICE uses the gathered observations of both types (contribution ob_e and completion ob_γ) to predict the probability value $P(e \in \gamma \ \cap \ \gamma \ completes)$, hence predicting $U_{e,\gamma}$. First, from both these observation types, hSPICE computes the utility of event e for the set of all possible states of PM γ (i.e., $\mathbb{S}_\gamma$) as follows:

$$U_{e,s} = \frac{|\{e : e \in \gamma_s \ \& \ \gamma \ completed\}|}{|\{e : e \otimes \gamma_s\}|} \tag{5}$$

where $U_{e,s} = P(e \in \gamma_s \ \cap \ \gamma \ completes)$. For event e of certain type T_e at certain position P_e within window w and for PM γ at certain state s, $U_{e,s}$ is computed as a ratio between the number of times PM γ *completes* and event e *contributes* to PM γ at state s (i.e., $e \in \gamma_s$) and the *total* number of times event e is processed with PM γ at state s (i.e., $e \otimes \gamma_s$).

Figure 4 shows the computed utility values $U_{e,s}$ from the observations shown in Table II. The values are shown as percentage values. The table shows the utility value of event e of type T_e at position P_e within a window for PMs at states s_0 and s_1. For example, in the table, event $e = A_2$ of type $T_e = A$ at position $P_e = 2$ within a window is processed with four PMs at state s_0 (PMs 3, 4, 5, and 6). However, it has contributed only to two PMs (3 and 4). Moreover, since only PM 3 completed, we account for the contribution of event $e = A_2$ only to PM 3. Therefore, in the table, the utility of event type $T_e = A$ at position $P_e = 2$ within a window for a PM at state s_0 equals to 25%, i.e., $U_{e,s_0} = \frac{1}{4} = 25\%$. The event type $T_e = A$ has never contributed to a PM at state s_1 since only the event type $T_e = B$ may contribute to a PM at state s_1. Therefore, the utility of an event of type $T_e = A$ at any position within a window for a PM at state s_1 is always zero as shown in the table. Similarly, the event type $T_e = B$ never contributes to a PM at state s_0. Hence, the utility of an event of type $T_e = B$ at any position within a window for a PM at state s_0 is always zero.

The utility values for all states of PM γ of pattern $q_i \in \mathbb{Q}$ together multiplied by the pattern weight w_{q_i} represent the predicted utility $U_{e,\gamma}$ of event e for PM $\gamma \subset q_i$, where $U_{e,\gamma_s} = f(T_e, P_e, s) = w_{q_i} * U_{e,s}$. Now, we need to store these predicted utility values $U_{e,\gamma}$ for all patterns (i.e., for $\mathbb{Q}$) so that, during load shedding, hSPICE can retrieve them. To reduce the storage overhead, in case of large window size, we use bins to group event utilities. Within window w, the utility values of event e of type T_e at several consecutive window positions (i.e., bin size bs) for PM γ_s at state s are grouped together by taking the average utility value of this event type T_e over all these positions for PM γ_s. For ease of presentation, we will use the bin of size $bs = 1$ if not otherwise stated. To efficiently retrieve the utility values during load shedding, we store the utilities in a table (called utility table UT) of three dimensions ($M \times N \times K$), where M represents the number of different event types, $N = \frac{ws}{bs}$, and K is the number of all possible states of all PMs of all patterns, i.e., $K = |\mathbb{S}_{\mathbb{\Gamma}}|$. Therefore, the storage overhead of the utility table UT is $O(M.N.|\mathbb{S}_{\mathbb{\Gamma}}|)$. Each cell $UT(T_e, P_e, S_\gamma)$ in the utility table stores the utility value $U_{e,\gamma}$ of event e of type T_e at position P_e within a window for PM γ at state S_γ, i.e., $U_{e,\gamma} = f(T_e, P_e, S_\gamma) = UT(T_e, P_e, S_\gamma)$. Hence, to get the utility $U_{e,\gamma}$ of event e for PM γ, hSPICE needs to perform only a single lookup in the utility table UT. This means that the time complexity to get $U_{e,\gamma}$ is $O(1)$ which considerably reduces the overhead of load shedding.

The input event stream might change over time, hence the predicted utilities of events for PMs might become inaccurate. One way to capture the changes in the input event stream and keep the event utility accurate is by periodically gathering statistics and recomputing the utility value $U_{e,\gamma}$.

3) Drop Amount: As we mentioned above, to maintain the given latency bound (LB) in an overload situation, we must drop ρ events from every window. However, hSPICE drops events from PMs, not from windows, where an event might be dropped from a PM while it is processed with another PM within the same window. Therefore, we must find a mapping between the number of events to drop per window (ρ) and the number of events to drop per PM within the window. To do that, let us first define the virtual window.

Virtual Window. The virtual window (vw) of window w is a set which contains triplets (e, s, O) consisting of event e of type T_e at position P_e within w, state $s \in \mathbb{S}_{\mathbb{\Gamma}}$, and the number of occurrences $O > 0$ which represents the number of times event e has been processed with a PM at state s within window w. More formally: $vw = \{(e, s, O) : \forall \ e \in w, \ \forall \ \gamma \in \mathbb{\Gamma}_w^T, \ O = |\{\gamma : e \otimes \gamma_s\}| > 0\}$. The virtual window vw of window w contains information on the number of times event e within window w is processed with each distinct state s of a PM in window w. The virtual window depends on the states of PMs in a window. Therefore, it is only possible to know the exact virtual window of window w when all events in window w are processed, i.e., when the set of all PMs $\mathbb{\Gamma}_w^T$ and their states in window w are known. However, we need to know the virtual window of window w before processing all events in window w since we use the virtual window to decide how many and which events must be dropped from PMs within window w.

Therefore, hSPICE predicts virtual window vw of window w by gathering statistics from the operator on already processed windows, denoted by W_{stat}. As mentioned above, in different windows, event distribution might be different (cf. Table I). Additionally, the occurrences of PM states at certain window positions might also be different in different windows. Hence, different windows might have different corresponding virtual windows. Therefore, to predict virtual window vw of window w, hSPICE first computes virtual window vw_j for each window w_j in the gathered statistics W_{stat}, where $j = 1, .., |W_{stat}|$. Then, hSPICE combines all triplets (e, s, O) from these virtual windows vw_j to construct the virtual window vw by taking the average value for the number of occurrence O of each triplet, i.e., $vw = \{(e, s, O) : e = e_j, s = s_j, O = O + \frac{O_j}{|W_{stat}|}, \forall\ (e_j, s_j, O_j) \in vw_j\}$. The size of virtual window vw (denoted by ws_v) is computed as the total number of occurrences of each triplet in vw as follows: $ws_v = \sum_{(e,s,O)\in vw} O$. The *virtual window size* represents the *number of times* events are processed with PMs in a window. Therefore, the average number of times (avg_O) an event is processed with a PM in window w is computed as follows: $avg_O = \frac{ws_v}{ws}$. For example, if every event is processed with two PMs within window w, then the virtual window size ws_v is twice the window size ws (i.e., $ws_v = 2.ws$) and $avg_O = 2$.

Dropping an event from window w implies that the event is dropped from the set of all current PMs $\mathbb{\Gamma}_w^P$ within window w. Therefore, if ρ events must be dropped from window w, it implies that, in total, $\rho_v \approx \rho * avg_O \approx \rho * \frac{ws_v}{ws}$ events must be dropped from all PMs $\mathbb{\Gamma}_w^T$ in window w (from virtual window vw of window w, as a shorthand). Hence, dropping ρ events from a window is similar to dropping ρ_v events from its virtual window. One approach to drop ρ_v events from a virtual window (i.e., ρ_v events in total from all PMs in a window) is to drop events equally (for example, equal percentage) from every PM in the window. However, not all PMs in a window have the same importance/same completion probability. Therefore, the drop amount per PM should take into consideration the importance of PMs in the window which in turn minimizes the adverse impact of dropping on QoR. Please note that it is not possible to get the utility of all events for all PMs in a window and then sort them. After that, drop those ρ_v events from PMs that have the lowest utilities. The reason for this is that the event utilities for PMs in a window are only known after processing all events in the window. This is because the event utilities depend on the current state of PMs ($\mathbb{\Gamma}_w^P$) in the window which is only known after processing the events in the window. Next, we explain how to drop the required number of events (ρ_v) from the virtual window of each window while considering the importance of PMs in the window.

Utility Threshold. The approach is to find a utility value (called utility threshold u_{th}) that is used as a threshold value to drop the needed amount of events from virtual window vw of window w. For each triplet (e, s, O) in virtual window vw, we get the utility value $u = U_{e,\gamma_s} = f(T_e, P_e, s)$ from the utility table UT. As the number of occurrences O in the triplet represents the number of times state s might occur at window position P_e, the number of occurrences O implies that the utility value $u = U_{e,\gamma_s}$ might occur O times in virtual window vw, denoted by the utility occurrences O_u for utility u, i.e, $O_u = O$. We accumulate the number of utility occurrences O_u for all utility values in vw in ascending order, denoted by the accumulative utility occurrences OC_u for the utility u, as follows: $OC_u = \sum_{u' \leq u} O'_u$. The accumulative utility occurrences OC_u for utility u means that there exist OC_u events in virtual window vw which have a utility value less or equal to the utility value u.

Therefore, using u as a threshold utility u_{th} enables hSPICE to drop OC_u events from PMs in a window. Hence, to drop ρ_v events from the virtual window, we must find a utility value $u = u_{th}$, where $OC_u = \rho_v$. To efficiently retrieve the utility threshold, we store the accumulative utility occurrences in an array (denoted by utility threshold array (UT_{th})) of the same size as the virtual window size ws_v as follows: $UT_{th}(i) = u$, where $i = 1, .., ws_v$ and $OC_u \geq i$ and $OC_u < OC_{u'}\ \forall\ u < u'$. Therefore, to drop ρ_v events from the virtual window, $u_{th} = UT_{th}(\rho_v)$. Hence, the time complexity to get u_{th} is $O(1)$. Please note that predicting the virtual window and building the utility threshold array are done during the model building task. While during the load shedding, hSPICE performs the following two tasks that have a time complexity of $O(1)$: 1) computing how many events to drop (i.e., ρ_v) per virtual window, and 2) determining what utility threshold (i.e., u_{th}) to use.

4) Load Shedding: In the above sections, we showed how to compute the utility of events for PMs within a window and how to predict the utility threshold. Now, we describe how hSPICE performs the load shedding, i.e., deciding whether an event should be dropped from a PM or not. Algorithm 1 clarifies how load shedding is performed.

For each event e within window w, before processing e with PM γ in window w, the operator asks the load shedder (LS) whether to drop event e from PM γ. If the LS returns True, the operator drops event e from PM γ, otherwise, it processes event e with PM γ. If there is no overload on the operator, there is no need to drop events and hence LS returns False which means that the operator can process event e with PM γ (cf. Algorithm 1, lines 2-3). On the other hand, if there is an overload on the operator, LS checks whether the utility $U_{e,\gamma}$ of event e for PM γ is higher than the utility threshold u_{th}. Therefore, the LS first gets the utility $U_{e,\gamma}$ of event e for PM γ from the utility table UT, where $U_{e,\gamma} = f(T_e, P_e, S_\gamma) = UT(T_e, P_e, S_\gamma)$. After that, hSPICE compares the utility value with the utility threshold u_{th}, where it returns True if $U_{e,\gamma} \leq u_{th}$, otherwise hSPICE returns False (cf. Algorithm 1, lines 4-7). This shows that hSPICE is lightweight in performing load shedding where the time complexity to decide whether or not to drop an event from a PM is $O(1)$.

Algorithm 1 Load shedder (PM granularity).

```
1: drop $(T_e, P_e, S_\gamma)$ begin
2:   if $!isOverloaded$ then    ▷ there is no overload hence no need to drop events
3:     return $False$
4:   else if $UT(T_e, P_e, S_\gamma) \leq u_{th}$ then
5:     return $True$
6:   else
7:     return $False$
8: end function
```

Having explained how to define the event utility, predict the event utility, find the utility threshold, and perform load shedding on the PM granularity, next, we describe how load shedding is performed on the window granularity.

B. Window Granularity

In the partial match granularity, as we showed above, for event e in window w, hSPICE must perform a check (lookup in UT) for every PM γ in w (i.e., for each $\gamma \in \mathbb{\Gamma}_w^P$) to decide whether or not to drop event e from PM γ. This implies that the time complexity to perform load shedding is $(|\mathbb{\Gamma}_{w}^{P}|.O(1))$ for every event within a window, where hSPICE must perform $|\mathbb{\Gamma}_{w}^{P}|$ lookups in UT. Although this shows that the overhead of performing load shedding in the PM granularity is low, in this section, we propose to perform load shedding on the window granularity which reduces the overhead of load shedding even further. The load shedding overhead for an event represents an additional latency which adds up to the processing latency of the event. Higher is the load shedding overhead for event e, higher is the processing latency of event e and hence lower is the operator throughput μ. This implies that reducing the load shedding overhead increases the operator throughput μ which in turn reduces the number of events that must be dropped to maintain LB, hence reducing the adverse impact of event shedding on QoR.

Performing load shedding on the window granularity implies that events are dropped from windows, i.e., in a window, an event is either dropped from all PMs or from none. This way, the load shedding is performed only once for every event in a window regardless of the number of current PMs $\mathbb{\Gamma}_{w}^{P}$ in the window which might considerably reduce the load shedding overhead. Of course, the event utility in the window granularity is less precise than the event utility in the PM granularity, which might adversely impact QoR. To drop events from a window, next, we introduce the event utility in a window, where, in overload cases, events with the lowest utilities are dropped from windows.

1) Event Utility: As mentioned above, an event in a window is processed with all current PMs $\mathbb{\Gamma}_{w}^{P}$ in the window. Therefore, the utility of event e in window w (denoted by $U_{e,w}$) depends on the utility of event e for all current PMs $\mathbb{\Gamma}_{w}^{P}$ in window w. We represent the utility $U_{e,w}$ of event e of type T_e at position P_e within window w as the sum of the utility of event e for all current PMs in window w, i.e., $\mathbb{\Gamma}_{w}^{P}$, as shown in Equation 6.

$$U_{e,w} = \sum_{\gamma \in \mathbb{\Gamma}_{w}^{P}} f(T_e, P_e, S_\gamma) \tag{6}$$

Computing $U_{e,w}$ as shown in this equation means that for each PM in a window, hSPICE must perform a lookup in the utility table UT, i.e., $|\mathbb{\Gamma}_{w}^{P}|$ lookups. However, this will result in the same overhead $(|\mathbb{\Gamma}_{w}^{P}|.O(1))$ as performing load shedding on the PM granularity.

To minimize this overhead, we must reduce the number of lookups in the utility table UT. To do that, we keep a summary on the distinct PM states and the number of occurrences of each distinct state in the window. In window w, at position P, multiple PMs might be at the same state. We define PM summary (denoted by SM_{w}^{P}) in window w at position P as a multiset that contains all distinct states of current PMs $\mathbb{\Gamma}_{w}^{P}$ at position P in window w and the number of occurrence of these PM states. Each element in PM summary is defined as a pair (s_k, O), where s_k represents a PM state and O represents the number of occurrences of state s_k in $\mathbb{\Gamma}_{w}^{P}$, i.e., $SM_{w}^{P}(s_k) = |\{\gamma : \gamma \in \mathbb{\Gamma}_{w}^{P}\ ,\ s_k = S_\gamma\}|$.

We use the PM summary SM_{w}^{P} to compute the utility $U_{e,w}$ of event e in window w as follows:

$$U_{e,w} = \sum_{S_\gamma \in SM_{w}^{P}} f(T_e, P_e, S_\gamma) * SM_{w}^{P}(S_\gamma) \tag{7}$$

For each distinct state of the current PMs ($\mathbb{\Gamma}_{w}^{P}$) in window w, hSPICE performs the lookup only once in the utility table UT to get the utility $U_{e,\gamma} = f(T_e, P_e, S_\gamma)$ of event e for PM γ. Then, hSPICE multiplies the utility $U_{e,\gamma}$ with the number of occurrences of state S_γ in w (i.e., $SM_{w}^{P}(S_\gamma)$). The event utility $U_{e,w}$ represents the sum of all multiplication results. Using Equation 7 might reduce the overhead of computing the utility $U_{e,w}$ considerably. This is because multiple PMs in a window might have the same state which means that the PM summary size might be much smaller than the number of PMs in a window, hence much less lookups in the utility table UT. This is more likely to happen if the number of states of all patterns is lower than the number of current PMs in a window, i.e., $|\mathbb{S}_{\mathbb{P}}| < |\mathbb{\Gamma}_{w}^{P}|$ where multiple PMs must be at the same state. The operator maintains the PM summary SM_{w}^{P} for each window w, where the PM summary is changed only if the state of PM $\gamma \in \mathbb{\Gamma}_{w}^{P}$ in window w changes, which does not happen frequently. Hence, maintaining the PM summaries for windows imposes only a small overhead on the operator.

2) Utility Threshold: As we mentioned above, to maintain the given latency bound (LB), the LS must drop ρ events from every window. To drop those ρ events from a window, similar to the PM granularity, we need to a find a utility threshold u_{th} in a window which enables the LS to drop those ρ events from a window. As in the PM granularity, we gather statistics on event distribution and on the distribution of PM summaries in the window. Then, we use these gathered statistics to compute the utility threshold u_{th}.

3) Load Shedding: Now, we describe the way hSPICE drops events from windows. Algorithm 2 clarifies how the load shedding is performed. Similar to dropping events from PMs, for each event e within window w, before processing event e with any PM in window w, the operator asks the LS whether or not to drop event e from window w. If LS returns True, the operator drops event e from window w, otherwise it processes event e with all current PMs $\mathbb{\Gamma}_{w}^{P}$ in window w.

If there is no overload on the operator, there is no need to drop events and hence LS returns False which means that the operator can process event e in window w (cf. Algorithm 2, lines 2-3). On the other hand, if there is overload on the operator, the LS checks whether the utility $U_{e,w}$ of event e in window w is higher than the utility threshold u_{th}, where the event must be dropped if $U_{e,w} \leq u_{th}$. To do that, the LS uses Equation 7 to compute the utility $U_{e,w}$. After that, LS compares the utility value $U_{e,w}$ with the utility threshold u_{th}, where it returns True if $U_{e,w} \leq u_{th}$, otherwise LS returns False (cf. Algorithm 2, lines 4-9). This shows that hSPICE performs load shedding for window granularity in the worst case in a time complexity of $(|\mathbb{\Gamma}_{w}^{P}|.O(1))$.

C. Supporting CEP Computational Models

So far, we have focused on using finite state machine [1], [11], [14], [17], [18] as a computational model to detect patterns. However, as we mentioned in Section II, there exist

Algorithm 2 Load shedder (window granularity).

```
applyLS $(T_e, P_e, SM_w^P)$ begin
  if $!isOverload$ then    ▷ there is no overload hence no need to drop events
    return $False$
  else
    $compute\ U_{e,w}\ using\ Equation\ 7$
    if $U_{e,w} \leq u_{th}$ then
      return $True$
    else
      return $False$
end function
```

several other computational models such as tree-based models [13], [19], [20], string-based models [21], and Petri Nets-based models [22]. In this section, we explain how our load shedding approach supports all the above computational models.

As we explained above, to assign a utility value to an event e, hSPICE (for both hSPICEPM and hSPICEW) depends on three features: 1) the current state of PM(s), 2) event type T_e, and 3) event position P_e in the window. Event type and event position in the window are independent of the computational model. Hence, to show that hSPICE supports other computational models, we must show that hSPICE is able to get PM states in these computational models, similar to the finite state machine model. To do that, let us first define a CEP pattern, a PM, and a PM state irrespective of the used computational model. In CEP, a pattern q is formed by using a set of events, event operators, and constraints [13]. For pattern q, a PM γ of pattern q represents an incomplete matching instance of pattern q, denoted by $\gamma \subset q$. For each event in pattern q, we assume that there is an assigned state that represents the state on a PM of pattern q. Additionally, there exists an initial state for pattern q. For example, in pattern $q = seq(A; B; C)$, we may assign state s_1 to event A, state s_2 to event B, and state s_3 to event C. In this example, a PM at state s_3 represents a complete match of pattern q, i.e., a complex event. Moreover, we may use state s_0 as the initial state of pattern q. Hence, a PM $\gamma \subset q$ starts at state s_0. If an event instance of event type A matches pattern q, the state of PM γ is updated to state s_1. Similarly, the state of PM γ changes to state s_2 or s_3 if an instance of event type B or C matches pattern q, respectively. Regardless of the used computational model, it is straightforward to assign states to PMs and update a PM state whenever an event matches the pattern and the PM progresses. Hence, hSPICE can support other computational models without any remarkable complexity.

IV. Performance Evaluations

In this section, we evaluate the performance of hSPICE, for both PM and window granularities, by using two real-world datasets and several representative queries.

A. Experimental Setup

Evaluation Platform. We run our evaluations on a machine that is equipped with 8 CPU cores (Intel 1.6 GHz) and a main memory of 24 GB. The OS used is CentOS 6.4. We run a CEP operator in a single thread on this machine, where this single thread is used as a resource limitation. Please note, the resource limitation can be any number of threads/cores and the behavior of hSPICE does not depend on a specific limitation. We implemented hSPICE by extending a prototype CEP framework that is implemented using Java.

Baseline. We compare the performance of hSPICE with three state-of-the-art load shedding strategies: 1) eSPICE: it is a black-box load shedding approach that drops events from windows [5]. 2) BL: we also implemented a black-box load shedding strategy (denoted by BL) similar to the one proposed in [9]. Additionally, it captures the notion of weighted sampling techniques in stream processing [23]. BL drops events from windows, where an event type (e.g., player ID or stock symbol) receives a higher utility proportional to its repetition in patterns and in windows. Then, depending on event type utilities, it uses uniform sampling to decide which event instances to drop from the same event type. 3) pSPICE: it is a white-box load shedding strategy that drops PMs [6].

Datasets. We use two real-world datasets. 1) A stock quote stream from the New York Stock Exchange, which contains real intra-day quotes of different stocks from NYSE collected over two months from Google Finance [24]. 2) A position data stream from a real-time locating system (denoted by RTLS) in a soccer game [25]. Players, balls, and referees are equipped with sensors that generate events containing their position, velocity, etc.

Queries. We apply five queries (Q_1, Q_2, Q_3, Q_4, and Q_5) that cover an important set of operators in CEP as shown in Table III: sequence operator, sequence operator with repetition (which also contains Kleene closure), disjunction operator, sequence with negation operator, and sequence with any operator [13], [14], [18], [20]. We use the *first* selection policy for all events in all queries. Additionally, we use the *consumed* consumption policy for the first event in all queries and the *zero* consumption policy for the rest events in all queries. Moreover, for all queries, we use time-based sliding window strategy.

In Table III, we use ws to refer to the window length. For stock queries (Q_1, Q_2, Q_3, and Q_4), C_i represents the stock quote of company i. Q_1 detects a complex event when rising or falling stock quotes of 10 certain stock symbols, by a given percentage, are detected within ws minutes in a certain sequence. Q_2 detects a complex event when 10 rising or 10 falling stock quotes of certain stock symbols *with repetition*, by a given percentage, are detected within ws minutes in a certain sequence. Q_3 detects a complex event if either Q_1 or Q_2 matches. Q_3 represents a multi-pattern operator. Q_4 is similar to Q_1 but it detects a complex event only if the stock quote of a certain company (i.e., C_5) does not change by a given percentage. Q_5 uses the RTLS dataset and it detects a complex event when any 3 defenders of a team (defined as `Di`) defend against a striker (defined as `S`) from the other team within ws seconds from the ball possessing event by the striker. The defending action is defined by a certain distance between the striker and the defenders. For this query, we use two strikers, one from each team.

Stock queries	
Q_1	**pattern seq**$(C_1; C_2; ..; C_{10})$ **where** *all* C_i *rise by* $x\%$ **or** *all* C_i *fall by* $x\%,\ i = 1..10$ **within** *ws* minutes
Q_2	**pattern seq**$(C_1; C_1; C_2; C_3; C_2; C_4; C_2; C_5; C_6; C_7; C_2; C_8; C_9; C_{10})$ **where** *all* C_i *rise by* $x\%$ **or** *all* C_i *fall by* $x\%,\ i = 1..10$ **within** *ws* minutes
Q_3	$\mathbf{Q_1 \vee Q_2}$
Q_4	**pattern seq**$(C_1; C_2; C_3; C_4; !\mathbf{C_5}; C_6; C_7; C_8; C_9; C_{10})$ **where** *all* C_i *rise by* $x\%$ **and** C_5 *does not rise by* $y\%$ **or** *all* C_i *fall by* $x\%$ **and** C_5 *does not fall by* $y\%$ $,\ i = 1..10$ *and* $i \neq 5$ **within** *ws* minutes
Soccer queries	
Q_5	**pattern seq**$(S; \mathbf{any}(3, D_1, D_2, .., D_n))$ **where** S *possesses ball* **and** $distance(S, D_i) \leq x\ meters$ $,\ i = 1..n$ and n is the number of players in a team **within** ws seconds

TABLE III: Queries.

B. Experimental Results

In this section, we evaluate the performance of hSPICE in comparison with other load shedding strategies. First, we show its impact on QoR, i.e., the number of false negatives and the number of false positives, using both strict and relaxed QoR. Then, we show how good hSPICE is in maintaining the given latency bound (LB). We refer to hSPICE when dropping events on window granularity as hSPICEW. While we refer to hSPICE when dropping events on PM granularity as hSPICEPM.

If not stated otherwise, we use the following settings. For all queries Q_1, Q_2, Q_3, Q_4, and Q_5, we use a *time-based* sliding window and a *time-based* predicate. We stream events to the operator from the datasets that are stored in files. We first stream events at input event rates which are less or equal to the operator throughput μ (maximum service rate) until the model is built. After that, we increase the input event rate to enforce load shedding as we will mention in the following experiments. The used latency bound $LB = 1$ second. We configure all load shedding strategies (i.e., hSPICE, eSPICE, BL, and pSPICE) to have a safety bound, where they start dropping events/PMs when the event queuing latency is greater than or equal to 80 % of LB, i.e., the safety bound equals to 200 milliseconds. We execute several runs for each experiment and show the mean value and standard deviation.

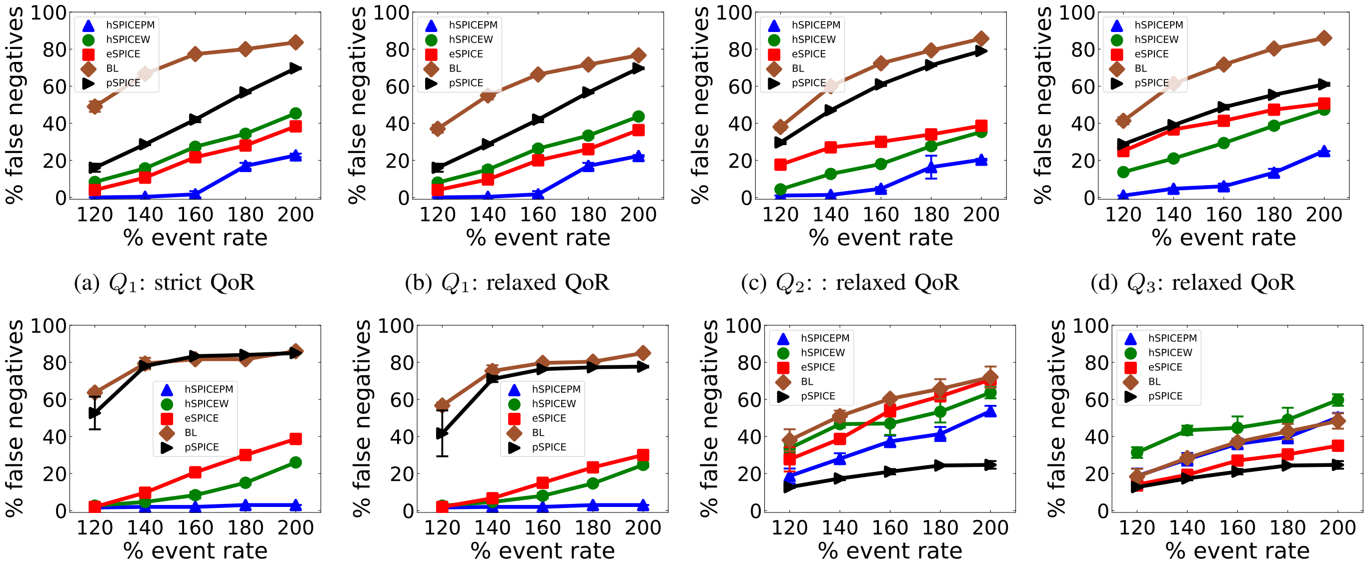
An important factor that might influence QoR is the input event rate. Higher is the input event rate, higher is the amount of events that must be dropped and hence higher is the impact of load shedding on QoR. Additionally, other factors that might impact QoR are the query properties, e.g., the used window size. Therefore, next, we show the impact of these factors on QoR, i.e., on false negatives and positives. Please note that in the case of using strict QoR, applying load shedding might result in false positives and false negatives for all queries (i.e., Q_1, Q_2, Q_3, Q_4, and Q_5). Additionally, when using relaxed QoR, applying load shedding might result in false negatives for all queries as well. However, it might result in false positives only in case of Q_4 since Q_4 has a negation operator. If the negated event is dropped by the load shedder, it might result in a false positive.

1) Impact of Event Rate on QoR: To evaluate the performance of hSPICE, we run experiments with queries Q_1, Q_2, Q_3, Q_4, and Q_5. To show the impact of input event rate, we stream both datasets to the operator with input event rates that are higher than the operator throughput μ by 20%, 40%, 60%, 80%, and 100% (i.e., event rate= 120%, 140%, 160%, 180%, and 200% of the operator throughput μ). Moreover, for Q_1, Q_2, Q_3 and Q_4, we use the following window sizes, respectively: 18, 35, 35, and 20 minutes. For Q_5, the used window size is 30 seconds. A new window is opened for Q_1, Q_2, Q_3, and Q_4 every 1 minute, i.e., the slide size is 1 minute. For Q_5, a new window is opened every 1 second. The average measured operator throughput μ (without load shedding) for queries Q_1, Q_2, Q_3, Q_4, and Q_5 are as follows: 23K, 14K, 8K, 36K, 27K events/second, respectively.

Impact on False Negatives. Figure 5 depicts the impact of event rates on false negatives for all queries. Figure 6 shows the ratio of dropped events or PMs (for pSPICE) with different event rates for Q_1 and Q_5. We observed similar results for Q_2, Q_3, and Q_4, hence we do not show them. In both figures, the x-axis represents the event rate. The y-axis in Figure 5 represents the percentage of false negatives while, in Figure 6, it represents the ratio of dropped events/PMs.

The percentage of false negatives might increase if the input event rate increases since more events/PMs must be dropped. Figure 5a and Figure 6a show the percentage of false negatives using strict QoR and the percentage of drop ratio for Q_1, respectively. As shown in Figure 5a, hSPICEPM has almost no impact on false negatives when the event rate is less or equal to 160% although hSPICEPM drops up to 80% of events when the event rate is 160% as depicted in Figure 6a. Increasing the event rate by more than 160% forces hSPICEPM to produce false negatives where the percentage of false negatives is 17% and 23% using event rates of 180% and 200%, respectively. The drop ratio starts to decrease when using a high event rate as shown in Figure 6a when using the event rate of 200%. The reason behind this is that when more events should be dropped, events with high utilities might be dropped. Dropping events with high utilities might hinder opening new PMs which in turn reduces the number of events that must be dropped. Since hSPICEPM drops more events compared to other load shedding strategies, i.e., eSPICE and BL, the impact of shedding in hSPICEPM on opening new PMs is higher which results in decreasing its drop ratio when the event rate is 200%. However, not opening those PMs might increase the number of false negatives.

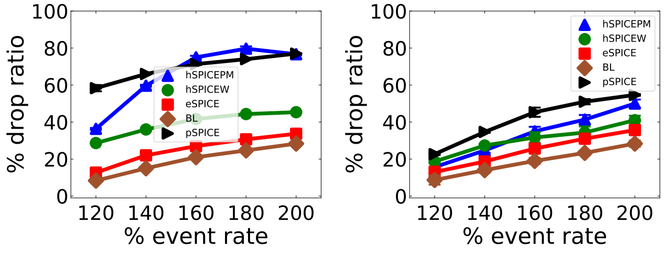
The percentage of false negatives caused by other load shedding strategies also increases when the event rate increases. As depicted in Figure 5a, when the event rate increases from 120% to 200%, the percentage of false negatives for hSPICEW, eSPICE, BL, and pSPICE increases from 8% to 45%, from 4% to 38%, from 48% to 84%, and from 16% to 70%, respectively. Moreover, the drop ratio increases with the event rate as shown in Figure 6a. hSPICEW performs, w.r.t. the percentage of false negatives, worse than hSPICEPM since hSPICEPM predicts the event utilities more accurately. Additionally, the used window size has a considerable impact on the performance of hSPICEPM. Please note that the used window sizes, in these experiments, are reasonable window sizes for the used datasets. However, if the window size



(a) Q_1: strict QoR (b) Q_1: relaxed QoR (c) Q_2: : relaxed QoR (d) Q_3: relaxed QoR

(e) Q_4: strict QoR (f) Q_4: relaxed QoR (g) Q_5: strict QoR (h) Q_5: relaxed QoR

Fig. 5: Impact of event rate on false negatives.



(a) Q_1 (b) Q_5

Fig. 6: Impact of event rate on drop ratio.

is much higher, which might be used in some applications, hSPICEW may perform better than hSPICEPM as we will show in Section IV-B2. The performance of hSPICEW is also worse than the performance of eSPICE as shown in Figure 5a. This is because hSPICEW drops more events than eSPICE as depicted in Figure 6a as the overhead of hSPICEW is higher than the overhead of eSPICE. The result shows that hSPICEPM significantly outperforms, w.r.t. the percentage of false negatives, all other load shedding strategies for Q_1 (sequence operator). Similar behavior is observed when using relaxed QoR as shown in Figure 5b.

Figures 5c shows results for Q_2 when using relaxed QoR. We observed similar behavior when using strict QoR, hence we do not show it. The figure shows that the performance, w.r.t. the percentage of false negatives, of all load shedders except eSPICE over Q_2 (sequence with repetition operator) is similar to their performance with Q_1 (sequence operator). The performance of eSPICE over Q_2 is worse than its performance over Q_1. The figure shows that the performance of hSPICEW is better than the performance of eSPICE over Q_2. However, the performance of hSPICEPM is, again, better than the performance of hSPICEW. The results show that hSPICEPM outperforms, w.r.t. the percentage of false negatives, all other load shedding strategies. The results for Q_3 (multi-pattern operator) are similar to the results for Q_2 as depicted in Figure 5d. The performance of hSPICEPM over Q_3 is, again, better than the performance of all other load shedding strategies. We observed similar results for Q_3 when using strict QoR.

Figure 5e and 5f depict the percentage of false negatives for Q_4 (sequence with negation operator) using strict and relaxed QoR, respectively. In Q_4, we limit the number of complex events to only one event per window, where the window is closed if a complex event is detected. We do that to determine the impact of the negation operator on the matching output. The performance of hSPICEPM, w.r.t. the percentage of false negatives, over Q_4 is considerably better than the performance of hSPICEPM over Q_1, Q_2, and Q_3. The reason behind this is that, in Q_4, there is at most one complex event per window in comparison to Q_1, Q_2, and Q_3 that detect all possible complex events in a window. Hence, in the case of Q_4, there exist many events in the window that have low utilities where dropping those events do not influence the percentage of false negatives. Figures 5e and 5f show that using hSPICEPM with different event rates introduces almost zero false negatives. The percentage of false negatives caused by using other load shedding strategies increases with increasing event rate. This shows that, for Q_4, hSPICEPM drastically reduces the percentage of false negatives compared to the other load shedding strategies.

Figures 5g and 5h show the percentage of false negatives for Q_5 (sequence with any operator) using strict and relaxed QoR, respectively. While Figure 6b shows the ratio of dropped events/PMs for Q_5. The drop ratio in Figure 6b increases when the event rate increases. However, the drop ratio of hSPICEPM and hSPICEW for Q_5 is lower than their drop ratio for Q_1. This is because the cost of processing events in Q_5 is higher than the cost of processing events in Q_1. Therefore, in Q_5, the overhead of performing load shedding in comparison to the event processing cost is lower which results in a low drop ratio. In Figures 5g and 5h, the percentage of false negatives caused by all load shedders increases when the input event rate increases.

Figure 5g shows that the performance of hSPICEPM is better than the performance of hSPICEW, eSPICE, and BL. However, pSPICE outperforms hSPICEPM. However, Figure 5h (i.e., using relaxed QoR) shows that hSPICEPM and hSPICEW perform almost worse than all other load shedding strategies. The reason behind this is that the impact of eSPICE and BL on the percentage of false negatives is reduced if there is no need to match the exact event instances (i.e., if the relaxed QoR is used). Moreover, the overhead of hSPICEPM and hSPICEW is high in comparison to other load shedding strategies. For every event in a window, hSPICEPM checks whether to drop the event or not from every individual PM within the window which increases the overhead of performing load shedding in hSPICEPM. Similarly, the overhead of hSPICEW is proportional to the number of PMs, as we discussed in Section III-B. While eSPICE and BL, for example, check whether to drop the event or not from the window regardless of the number of PMs within the window which reduces the overhead of performing load shedding in these approaches. The overhead of hSPICEPM and hSPICEW is high in all queries, however, the overhead impact is worse in Q_5. This is because in Q_5 the utility values are spread and less accurately predicted since Q_5 represents an *any* operator in comparison to other queries that use a *sequence* operator. Q_5 matches an event of any type (any player) with a PM at any state, unlike the *sequence* operator that matches only an event of a certain type with a PM at a certain state. Hence, in the case of Q_5, the majority of events in a window have similar utilities for all PM states.

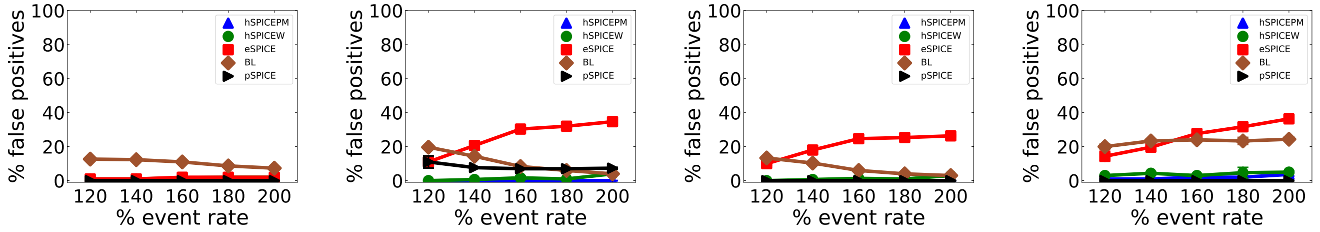
Impact on False Positives. As we mentioned above, for all queries, dropping events might result in false positives when using strict QoR. However, for only Q_4 (sequence with negation operator), dropping events might result in false positives in the case of using relaxed QoR. Please recall that Q_4 detects at most one complex event per window. Figure 7 depicts the percentage of false positives with different event rates for queries Q_1, Q_4, and Q_5. We observed similar results for Q_2 and Q_3, hence we do not show them. In the figure, the x-axis represents the event rate and the y-axis represents the percentage of false positives. Figure 7 shows that hSPICEPM and hSPICEW perform very well with all queries where the percentage of false positives caused by both hSPICEPM and hSPICEW is almost zero for different event rates.

The percentage of false positives caused by eSPICE in the case of Q_1 is negligible as depicted in Figure 7a. While the percentage of false positives caused by eSPICE increases with increasing the event rate for Q_4 and Q_5. Figure 7 shows that, for the majority of queries, the percentage of false positives produced when using BL decreases when increasing the event rate. The reason behind this is that, for low event rates, BL needs to drop fewer events, and hence more redundant events might exist in windows that might match the pattern. On the other hand, with a high event rate, BL must drop more events which makes it hard to have redundant events that might match the pattern. Higher is the probability to match the pattern, higher is the probability to get false positives. pSPICE drops PMs, therefore, it might results in false positives only if the strict QoR is used. The percentage of false positives caused by pSPICE in the case of Q_1 and Q_5 is negligible as depicted in Figures 7a and 7d. While the percentage of false positives caused by pSPICE slightly decreases with increasing the event rate for Q_4, using strict QoR.

2) Impact of Window Size on QoR: In this section, we analyze the impact of window size on QoR. A very large window might result in a large utility table (UT) that does not fit into the cache memory, and hence the lookup time in UT might increase. This results in increasing the load shedding overhead of hSPICEPM, hence dropping more events (i.e., adversely impact QoR). Moreover, using a very large window might increase the number of concurrent PMs $\mathbb{\Gamma}_w^P$ in the window, hence, also, increasing the load shedding overhead of hSPICEPM. A large utility table UT and a high number of concurrent PMs $\mathbb{\Gamma}_w^P$ might increase the load shedding overhead of hSPICEW as well. However, a high number of concurrent PMs $\mathbb{\Gamma}_w^P$ implies that there exist many PMs at the same state, hence hSPICEW needs to perform only few lookups in UT compared to hSPICEPM since hSPICEW performs a lookup in UT only once for each distinct PM state (cf. Section III-B). This implies that the overhead of hSPICEW for large windows might be much lower than the overhead of hSPICEPM, hence the impact of hSPICEW on QoR using large windows might be much lower than the impact of hSPICEPM on QoR. Please note that we may reduce the size of UT by using bins as we discussed in Section III. However, there still exist situations where the utility table UT might be large since bins can help only in the case of very large window sizes. For example, if the number of event types is high, the size of UT might also be large.

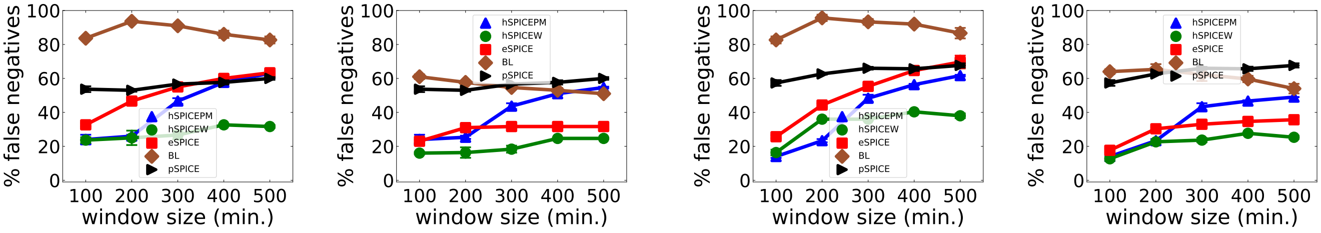
To show the impact of window size on QoR, we run experiments with queries Q_1 and Q_2 where we use a fixed event rate of 180%, i.e., the input event rate is higher than the operator throughput μ by 80%. To show the impact of window sizes, we vary the window size for both Q_1 and Q_2. The used window sizes for Q_1 and Q_2 are as follows: 100, 200, 300, 400, and 500 minutes. A new window is opened for Q_1 and Q_2 every 5 minutes, i.e., the slide size is 5 minutes. Figure 8 and Figure 9 depict the results for both queries. In both figures, the x-axis represents the event rate. The y-axis in Figure 8 represents the percentage of false negatives while the y-axis in Figure 9 represents the percentage of positives. We observed similar results for Q_3, Q_4, and Q_5, hence we do not show them.

Figure 8a depicts the percentage of false negatives for Q_1 using strict QoR. The figure shows that for the window sizes 100 and 200 minutes, the performance, w.r.t. the percentage of false negatives, of hSPICEPM is similar to the performance of hSPICEW. However, hSPICEPM still performs better than other load shedding strategies (i.e., eSPICE, BL, and pSPICE). For very large window sizes, the performance of hSPICEPM might become worse due to the following reasons. Increasing



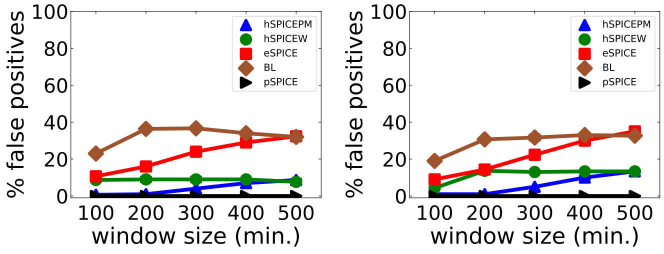
(a) Q_1: strict QoR (b) Q_4: : strict QoR (c) Q_4: relaxed QoR (d) Q_5: strict QoR

Fig. 7: Impact of event rate on false positives.



(a) Q_1: strict QoR (b) Q_1: relaxed QoR (c) Q_2: strict QoR (d) Q_2: relaxed QoR

Fig. 8: Impact of window size on false negatives.



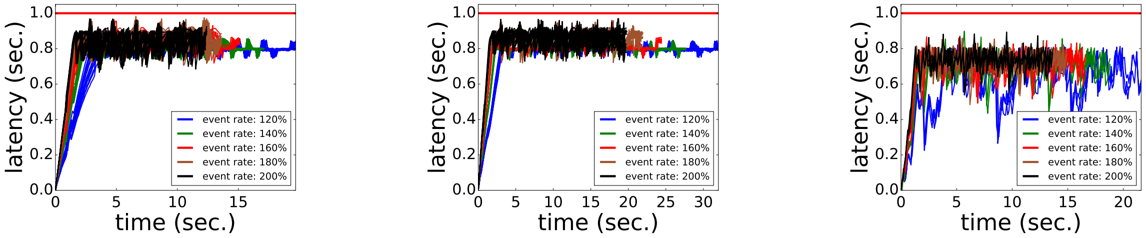
(a) Q_1: strict QoR (b) Q_2: strict QoR

Fig. 9: Impact of window size on false positives.

the window size might result in increasing the completion probability of PMs within the window. This implies that more events in the window might acquire a high utility value. Therefore, in this case, the load shedding impact on QoR might increase. Moreover, increasing the window size might increase the number of concurrent PMs within the window where more PMs might open. This implies that the overhead of load shedding of hSPICEPM might increase with increasing the window size since its overhead is proportional to the number of PMs in windows. This might result in dropping more events, hence increasing the impact on QoR. This is observed in Figure 8a where the percentage of false negatives caused by hSPICEPM increases when the window size increases. The figure shows that for large window sizes (i.e., 400 and 500 minutes), the performance, w.r.t. the percentage of false negatives, of hSPICEPM is similar to the performance of eSPICE and pSPICE. However, hSPICEW considerably outperforms hSPICEPM when the window size is larger than 200 minutes as depicted in the figure. The percentage of false negatives caused by hSPICEW slightly increases when the input event rate increases. The percentage of false negatives caused by eSPICE, also, increases with increasing the window size as shown in the figure. The results for pSPICE are also similar. The results for BL shows that the percentage of false negatives is only slightly increasing when increasing the window size to 200 minutes after that it starts to decrease. This shows that hSPICEW performs, w.r.t. the percentage of false negatives, very well with relatively large window sizes and it outperforms eSPICE, BL, and pSPICE regardless of the used window size.

In the case of using relaxed QoR for Q_1, hSPICEPM, hSPICEW, and pSPICE produce similar results to the results when using strict QoR as depicted in Figure 8b. However, the percentage of false negatives caused by eSPICE and BL decreases compared to the case when using strict QoR. The figure shows that for a window size longer than 300 minutes, eSPICE outperforms hSPICEPM. The performance of hSPICEW is, again, better than the performance of hSPICEPM, eSPICE, BL, and pSPICE regardless of the used window size as depicted in the figure. The results for Q_2 show similar behavior as depicted in Figures 8c and 8d where hSPICEW performs very well regardless of the used window size. Figures 9a and 9b depict the percentage of false positives for Q_1 and Q_2, respectively. The figures show that the percentage of false positives caused by hSPICEPM is only slightly increasing when the window size increases, while the percentage of false positives caused by hSPICEW is almost same with different window sizes. On the other hand, the percentage of false positives caused by eSPICE and BL increases with increasing the window size. pSPICE results in almost zero false positives for both Q_1 and Q_2.

3) Maintaining Latency Bound: The main objective of hSPICE is to minimize the degradation in QoR while maintaining a given latency bound (LB). As mentioned above, LB is 1 second and hSPICE drops events when the event queuing latency is greater than or equal to 80% of LB (i.e.,



(a) Q1

(b) Q2

(c) Q5

Fig. 10: Maintaining latency bound.

800 milliseconds). The event rate is an important factor that influences the ability of hSPICE to maintain LB. Therefore, in this section, we show the ability of hSPICE to maintain the given latency bound (LB) with different event rates. Figure 10 shows the event latency for Q_1, Q_2, and Q_5 where the event latency is the sum of the event queuing latency and the event processing latency. The event latency depicted in Figure 10 is measured when evaluating those three queries using the same settings as in Section IV-B1. In the figure, the x-axis represents the event rate and the y-axis represents the induced event latency. We observed similar results for Q_3 and Q_4 and all other queries when using different settings (e.g., using different window sizes), hence we do not show them.

Figures 10a, 10b, and 10c depict results for Q_1, Q_2, and Q_5, respectively. The figures show that hSPICE always maintains the given latency bound irrespective of the event rate. In the figure, the induced event latency stays around 800 milliseconds (i.e., 80% of LB which is used to have a safety bound). As can be seen, the objective of maintaining the latency bound is successfully achieved by hSPICE.

4) Discussion: hSPICE shows its ability to maintain the given latency bound while minimizing the degradation in QoR. Through extensive evaluations, we show that hSPICE outperforms, w.r.t. QoR, eSPICE, BL, and pSPICE for the majority of queries– especially for *sequence* operators. The performance of hSPICE for the *any* operator is worse than the performance of other load shedding strategies when using relaxed QoR. We also show that significantly increasing the window size might increase the impact of hSPICEPM on QoR. In short, we show that hSPICEPM has a considerably good performance, w.r.t. QoR, in the case of reasonable window sizes. Whereas hSPICEW is only slightly influenced by the increased size of the windows. Hence, depending on the window size requirement of the application, either hSPICEPM or hSPICEW might be used to minimize the load shedding impact on QoR.

V. Related Work

Complex event processing (CEP) systems are used in many applications to detect interesting patterns in input event streams [1], [2], [3], [26]. There exist several well-defined event patterns in CEP (also called event operators), e.g., sequence, negation, any, disjunction, and conjunction [13], [18], [27]. In CEP systems, the input event stream is continuous and may have a high volume. Moreover, the events are usually required to be processed in near real-time [7], [8]. Therefore, in CEP, there exist several techniques aiming to process the input events in a given latency bound such as parallelism, optimizations, and pattern sharing [1], [2], [17], [18]. However, these techniques are not always sufficient or even possible, therefore, researchers propose to use load shedding.

Recently, there have been several works on load shedding in CEP [5], [6], [9], [10]. All these approaches aim to minimize the impact of load shedding on QoR. The approaches in [5], [9] propose to drop events with the lowest utility from a CEP operator while the work in [6] drops PMs with the lowest utility in overload situations. In [9], the utility of an event depends on the event type and its frequency in the input event stream. While in [5] the utility of an event depends on the event type and its position in the window. In [6], the utility of a PM depends on its completion probability and its estimated processing cost. To predict the utility of a PM, the authors propose to use as learning features the current state of the PM and the remaining events in the window. Unlike all these approaches, our approach drops events from PMs where an event might have different importance for different PMs. As a result, our approach predicts the event utilities more accurately and performs dropping more precisely, thus reducing the adverse impact of load shedding on QoR.

In [10], the authors propose a load shedding approach to drop PMs and events. They assign utilities to PMs in a similar way to [6], i.e., depending on the completion probability of PMs and their estimated processing cost. When load shedding is triggered, the approach performs the following: 1) it selects a set of PMs (called PM shedding set) with the lowest utilities and adds all events that belong to PMs in the PM shedding set to an event shedding set (denoted by E_D). 2) It first drops all PMs in the PM shedding set. Then, it drops incoming events e that belong to the event shedding set from all PMs, i.e., if $e \in E_D$, drop e. The event dropping stops when the given latency bound is not violated anymore. The approach assumes that events that are part of low utility PMs have low importance and can be dropped with a low impact on QoR. However, this is not necessarily true as a PM with low utility may also contain highly important events. This might result in dropping important events. Furthermore, as this load shedding approach depends only on PMs to build the event shedding set, this implies that different events in a pattern have different probabilities to be chosen for the event shedding set. Moreover, this load shedding approach uses event content to check if an event belongs to the event shedding set. However, if events contain floating point, text, or image content, it is hard to find an exact match with events in the event shedding set. Hence, in these cases, it is not clear if this approach could maintain the given latency. Additionally, using events with their content in the event shedding set might considerably increase the load shedding overhead. The load shedding overhead in hSPICE, on the other hand, is independent of the event content. Furthermore, the load shedding approach in [10] seems to only support skip-till-any-match semantic [11] which represents a small set of known pattern semantics in CEP [13], [14], [15]. Moreover, this approach does not support the negation operator. In contrast, hSPICE supports all commonly used event operators and selection and consumption policies.

In the domain of approximate CEP, the authors in [28] propose a white-box approach (called RC-ACEP) to drop events from PMs in overload cases. The approach aims to minimize the degradation in QoR. They assign utilities to PMs depending on completion probabilities of the PMs– higher is the completion probability, higher is the utility. The idea is to process input events firstly with PMs that have the highest utilities. For each newly coming input event, RC-ACEP stops processing the previous event, recalculates and sorts PM utilities, and then processes the new events with the sorted PMs. However, recalculating and sorting PM utilities for every input event imposes a high overhead. Moreover, they do not consider the importance of input events for PMs where input events might have different importance for different PMs.

Various approximation techniques are frequently used to avoid resource constraints in various domains such as distributed graph processing [29], in-network processing [30], [31], stream processing [4], [23], [32], etc. Load shedding has, especially, been extensively studied in the stream processing domain [4], [7], [16], [23], [32], [33], [34], [35]. In [4], [23], [34], the authors assume that the importance of a tuple depends on the tuple's content. [23] assumes the mapping between the utility and tuple's content is given, for example, by an application expert, while [4], [23] learn this mapping online depending on the used query. The authors in [32] assume that the importance of a tuple depends on the processing latency of the tuple– higher is the processing latency of a tuple, lower is its importance. Therefore, they drop those tuples that have the highest processing latencies. In [7], the authors fairly select tuples to drop from different input streams by combining two techniques: stratified sampling and reservoir sampling. The authors in [35] also propose to use stratified sampling and reservoir sampling to perform the approximate join. In both these papers, the authors assume that tuples have the same utility values and impose the same processing latency. All these works do not capture the correlation between events in patterns which is important in CEP. For example, if the pattern is $seq(A; B)$, then events of type A are only important if the stream contains events of type B and vise-versa. Our approach implicitly captures this correlation.

VI. Conclusion

In this paper, we proposed an efficient, lightweight load shedding strategy called hSPICE which combines the advantages of both black-box and white-box state-of-the-art load shedding strategies. hSPICE consists of two load shedding approaches hSPICEPM and hSPICEW. hSPICEPM drops events from PMs within windows, while hSPICEW drops events from windows. In overload cases, hSPICE drops events from partial matches (i.e., using hSPICEPM) or from windows (i.e., using hSPICEW) to maintain a given latency bound. To assign a utility value to an event for a partial match, hSPICE uses three features: 1) event type, 2) event position in the window, and 3) the current state of the partial match. By using a probabilistic model, hSPICE uses these features to predict the event utility. Through extensive evaluations on two real-world datasets and several representative queries, we show that, for the majority of queries, hSPICE outperforms, w.r.t. QoR, state-of-the-art load shedding strategies. Moreover, we show that hSPICE always maintains the given latency bound regardless of the incoming input event rate.

Acknowledgement

This work was supported by the German Research Foundation (DFG) under the research grant "PRECEPT II" (BH 154/1-2 and RO 1086/19-2). The authors would like to thank Nabila Hashad for helping with implementation.

References

[1] R. Mayer, A. Slo, M. A. Tariq, K. Rothermel, M. Gräber, and U. Ramachandran, "Spectre: Supporting consumption policies in window-based parallel complex event processing," in *Proc. of the 18th ACM/IFIP/USENIX Middleware Conf.*, 2017.

[2] C. Balkesen, N. Dindar, M. Wetter, and N. Tatbul, "Rip: Run-based intra-query parallelism for scalable complex event processing," in *Proc. of the 7th ACM DEBS Conf. on Distributed Event-based Systems*, 2013.

[3] N. Zacheilas, V. Kalogeraki, N. Zygouras, N. Panagiotou, and D. Gunopulos, "Elastic complex event processing exploiting prediction," in *IEEE Int. Conf. on Big Data*, 2015.

[4] C. Olston, J. Jiang, and J. Widom, "Adaptive filters for continuous queries over distributed data streams," in *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, 2003.

[5] A. Slo, S. Bhowmik, and K. Rothermel, "espice: Probabilistic load shedding from input event streams in complex event processing," in *Proceedings of the 20th International Middleware Conference*, ser. Middleware '19. ACM, 2019.

[6] A. Slo, S. Bhowmik, A. Flaig, and K. Rothermel, "pspice: Partial match shedding for complex event processing," in *IEEE BigData 2019*.

[7] D. L. Quoc, R. Chen, P. Bhatotia, C. Fetzer, V. Hilt, and T. Strufe, "Streamapprox: Approximate computing for stream analytics," in *Proc. of the 18th ACM/IFIP/USENIX Middleware Conf.*, 2017.

[8] H. Röger, S. Bhowmik, and K. Rothermel, "Combining it all: Cost minimal and low-latency stream processing across distributed heterogeneous infrastructures," in *Proceedings of the 20th International Middleware Conference*, ser. Middleware '19, 2019.

[9] Y. He, S. Barman, and J. F. Naughton, "On load shedding in complex event processing," in *ICDT*, 2014.

[10] B. Zhao, N. Q. Viet Hung, and M. Weidlich, "Load shedding for complex event processing: Input-based and state-based techniques," in *ICDE 2020*.

[11] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman, "Efficient pattern matching over event streams," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 147160.

[12] A. Slo, S. Bhowmik, and K. Rothermel, "Hspice: State-aware event shedding in complex event processing," in *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*, ser. DEBS 20. New York, NY, USA: ACM, 2020, p. 109120.

[13] S. Chakravarthy and D. Mishra, "Snoop: An expressive event specification language for active databases," *Data Knowl. Eng.*, vol. 14, no. 1, pp. 1–26, Nov. 1994.

[14] G. Cugola and A. Margara, “Tesla: A formally defined event specification language,” in *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, ser. DEBS ’10. New York, NY, USA: ACM, 2010, pp. 50–61.

[15] D. Zimmer, “On the semantics of complex events in active database management systems,” in *Proceedings of the 15th International Conference on Data Engineering*, ser. ICDE ’99. Washington, DC, USA: IEEE Computer Society, 1999, pp. 392–.

[16] N. Tatbul and S. Zdonik, “Window-aware load shedding for aggregation queries over data streams,” in *Proc. of the 32nd Int. Conf. on Very Large Data Bases*, 2006.

[17] M. Ray, C. Lei, and E. A. Rundensteiner, “Scalable pattern sharing on event streams,” in *Proc. of the Int. Conf. on Management of Data*, 2016.

[18] E. Wu, Y. Diao, and S. Rizvi, “High-performance complex event processing over streams,” in *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, 2006.

[19] S. Chakravarthy, V. Krishnaprasad, E. Anwar, and S.-K. Kim, “Composite events for active databases: Semantics, contexts and detection,” in *Proceedings of the 20th International Conference on Very Large Data Bases*, ser. VLDB ’94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, p. 606617.

[20] Y. Mei and S. Madden, “Zstream: a cost-based query processor for adaptively detecting composite events,” in *SIGMOD Conference*, 2009.

[21] R. Sadri, C. Zaniolo, A. Zarkesh, and J. Adibi, “Expressing and optimizing sequence queries in database systems,” *ACM Trans. Database Syst.*, vol. 29, no. 2, p. 282318, Jun. 2004.

[22] S. Gatziu and K. R. Dittrich, “Events in an active object-oriented database system,” in *Rules in Database Systems*, N. W. Paton and M. H. Williams, Eds. London: Springer London, 1994, pp. 23–39.

[23] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker, “Load shedding in a data stream manager,” in *Proc. of the 29th Int. Conf. on Very Large Data Bases*, 2003.

[24] “Google Finance,” https://www.google.com/finance, 05.05.2019.

[25] DEBS 2013. Accessed: 2019-08-16. [Online]. Available: https://debs.org/grand-challenges/2013/

[26] G. F. Lima, A. Slo, S. Bhowmik, M. Endler, and K. Rothermel, “Skipping unused events to speed up rollback-recovery in distributed data-parallel cep,” in *2018 IEEE/ACM 5th International Conference on Big Data Computing Applications and Technologies (BDCAT)*, Dec 2018, pp. 31–40.

[27] M. Liu, M. Li, D. Golovnya, E. A. Rundensteiner, and K. Claypool, “Sequence pattern query processing over out-of-order event streams,” in *IEEE 25th Int. Conf. on Data Engineering*, 2009.

[28] Z. Li and T. Ge, “History is a mirror to the future: Best-effort approximate complex event matching with insufficient resources,” *Proc. VLDB Endow.*, vol. 10, no. 4, pp. 397–408, Nov. 2016.

[29] Z. Shang and J. X. Yu, “Auto-approximation of graph computing,” *Proceedings of the VLDB Endowment*, vol. 7, no. 14, pp. 1833–1844, 2014.

[30] S. Bhowmik, M. A. Tariq, J. Grunert, D. Srinivasan, and K. Rothermel, “Expressive content-based routing in software-defined networks,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 11, pp. 2460–2477, Nov 2018.

[31] S. Bhowmik, M. A. Tariq, A. Balogh, and K. Rothermel, “Addressing TCAM limitations of software-defined networks for content-based routing,” in *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems (DEBS)*, 2017.

[32] N. Rivetti, Y. Busnel, and L. Querzoni, “Load-aware shedding in stream processing systems,” in *Proc. of the 10th ACM Int. Conf. on Distributed and Event-based Systems*, 2016.

[33] E. Kalyvianaki, M. Fiscato, T. Salonidis, and P. Pietzuch, “Themis: Fairness in federated stream processing under overload,” in *Proc. of the Int. Conf. on Management of Data*, 2016.

[34] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis, “Concept-driven load shedding: Reducing size and error of voluminous and variable data streams,” in *IEEE Int. Conf. on Big Data*, 2018.

[35] W. H. Tok, S. Bressan, and M.-L. Lee, “A stratified approach to progressive approximate joins,” in *Proc. of the Int. Conf. on Extending Database Technology: Advances in Database Technology*, 2008.

Ahmad Slo received the BS degree in computer science from Aleppo University, Aleppo, Syria, and the MS degree in computer and communications systems engineering from the Technical University of Braunschweig, Braunschweig, Germany. He is currently working toward the Ph.D. degree at the University of Stuttgart, Stuttgart, Germany. He is currently with the Distributed Systems Research Group, University of Stuttgart. His research interests include complex event processing, stream processing, load shedding, and low latency event processing.

Sukanya Bhowmik received her doctoral degree from University of Stuttgart, Germany, in 2017. She is currently working as a postdoctoral researcher at the Distributed Systems research group of University of Stuttgart. Her research interests include stream/-complex event processing, high performance communication middleware, in-network event processing, software-defined networking, and distributed graph processing, with a focus on scalability, line-rate performance, resource efficiency, and adaptability aspects.

Kurt Rothermel received his doctoral degree in Computer Science from University of Stuttgart in 1985. From 1986 to 1987 he was a Post-Doctoral Fellow at IBM Almaden Research Center in San José, U.S.A., and then joined IBM’s European Networking Center in Heidelberg. Since 1990 he is a Professor for Computer Science at the University of Stuttgart. From 2003 to 2011 he was head of the Collaborative Research Center Nexus (SFB 627), conducting research in the area of mobile context-aware systems. His current research interests are in the field of distributed systems, computer networks, and mobile systems.