eSPICE: Probabilistic Load Shedding from Input Event Streams in Complex Event Processing

Ahmad Slo, Sukanya Bhowmik, Kurt Rothermel
University of Stuttgart
firstName.lastName@ipvs.uni-stuttgart.de

Abstract

Complex event processing systems process the input event streams on-the-fly. Since input event rate could overshoot the system's capabilities and results in violating a defined latency bound, load shedding is used to drop a portion of the input event streams. The crucial question here is how many and which events to drop so the defined latency bound is maintained and the degradation in the quality of results is minimized. In stream processing domain, different load shedding strategies have been proposed but they mainly depend on the importance of individual tuples (events). However, as complex event processing systems perform pattern detection, the importance of events is also influenced by other events in the same pattern. In this paper, we propose a load shedding framework called eSPICE for complex event processing systems. eSPICE depends on building a probabilistic model that learns about the importance of events in a window. The position of an event in a window and its type are used as features to build the model. Further, we provide algorithms to decide when to start dropping events and how many events to drop. Moreover, we extensively evaluate the performance of eSPICE on two real-world datasets.

CCS Concepts • **Information systems** → *Data streams*; *Stream management*; • **Theory of computation** → *Streaming models*;

Keywords Complex Event Processing, Stream Processing, Load Shedding, Approximate Computing, latency bound, QoS

ACM Reference Format:
Ahmad Slo, Sukanya Bhowmik, Kurt Rothermel. 2019. eSPICE: Probabilistic Load Shedding from Input Event Streams in Complex Event Processing. In *Middleware '19: Middleware '19: 20th International Middleware Conference, December 8–13, 2019, Davis, CA, USA.* ACM, UC DAVIS, CA, 13 pages. https://doi.org/10.1145/3361525.3361548

1 Introduction

Complex event processing (CEP) is an efficient and scalable paradigm for processing event streams. An operator in a CEP system is used to detect important situations by analyzing the input event streams. It performs pattern matching by correlating the events (called primitive events) from the input event streams and generates, as output, complex events which represent the occurrence of specific situations, e.g., fire detection, stock changes, intrusion detection, etc. [10, 17, 21]. In CEP [1, 4, 7, 20, 24], the input event stream is partitioned into independent windows of events where a window captures temporal relations between events. Windows might overlap and hence an event can be part of several windows.

In CEP, the volume of input event streams is huge and cannot be processed on a single machine. Therefore, distribution and parallelism are frequently used in CEP, where the CEP operator graph is distributed on multiple compute nodes. Moreover, each CEP operator runs on one or more compute nodes [4, 7, 15, 17, 20]. The underlying assumption of the above works is that there are infinite available resources, e.g., in cloud. However, there are various reasons for considering limited resources such as: 1) limited monetary budget, and 2) limited compute resources if operators run in private clouds due to security or response time reasons.

In most CEP applications, the detected complex events are useless if they are not detected within a certain latency bound [7, 22]. Moreover, many applications accept a reduced result quality, e.g., network monitoring, traffic monitoring, stock market [4, 21, 32]. To avoid violating a defined latency bound or crashing the system in the face of high incoming event rate and limited resources, load shedding may be necessary. Load shedding drops events from input event streams of an operator, thereby reducing its load. However, dropping events might adversely impact the quality of the CEP output where important situations could be missed or falsely detected in the input event stream. Thus, it is crucial to drop those events that have less impact (low utility/importance) on the quality of results.

There are primarily three challenges facing the decision to drop events in CEP systems: 1) Deciding on which events to drop since the utility of an event depends on multiple factors, e.g., other events in the pattern, on the order of events in the pattern, and on the input event stream. 2) Calculating the number of events to drop in order to maintain a given latency bound since an event may be dropped from some windows while it is still there in other windows. 3) Dropping events in an efficient way to reduce the overhead of load shedding.

Load shedding has been proposed by several research groups [13, 14, 21, 23, 29, 30] in the stream processing domain. They focus mainly on individual events, where each event has an associated utility value that reflects its importance for the result quality. However, since CEP systems perform pattern matching, the utility of events is also influenced by other events in the same pattern. For example, let the pattern be $\{seq(A;B) \vee seq(A;C)\}$. It is clear that events of type A are more important than events of type B and C. However, if all events of type B and C are dropped, then no complex events would be detected. Hence, we cannot consider only the utility of each event individually but we must also take into consideration other events in the pattern and in the input event streams. So far, there is only little work on load shedding in CEP. In [12], the authors consider the dependency between different events of the same pattern. However, they do not consider the order of events in patterns and input event streams which is extremely important in CEP such as in the sequence operator.

In this paper, we propose a load shedding framework, called eSPICE, for CEP systems. eSPICE is efficient and lightweight. Moreover, it considers the dependency between events of the same pattern as well as the order of events in the pattern and in input event streams. In addition, it also considers the impact of the same event residing in overlapping windows on the quality of results, where the same event may be in different positions in different windows. To capture the utility of an event in different windows, we design a probabilistic model that uses the **relative position** of events in a window and their **types** as learning features. The goal of our load shedding strategy is to maintain a given latency bound while minimizing the adverse impact of dropping events on the quality of results. More specifically, our contributions in this paper are as follows:

- We propose an efficient lightweight load shedding strategy that uses a probabilistic model to capture the importance of events in a window. The importance of an event is influenced by its type and its relative position in a window. The idea behind this approach is that the events, in specific positions within a window, that contribute to building a complex event in one window are more likely to build complex events in other windows as well.
- We provide an algorithm to estimate the number of events to drop in order to maintain the given latency bound. It also estimates the intervals within which the drop should take place.
- In order to show the effectiveness of our proposed load shedding strategy under realistic settings, we implement and thoroughly evaluate eSPICE for a broad range of CEP operators using real world datasets.

The rest of the paper is structured as follows. Section 2 presents the background for this work and formally states the problem that we address. In Section 3, we provide a detailed explanation of how the different components of eSPICE are used for our probabilistic load shedding strategy. Section 4 presents results obtained from extensively evaluating eSPICE. Finally, we discuss the related work and the conclusion in Sections 5 and 6, respectively.

2 Preliminaries and Problem Statement

In complex event processing (CEP) systems, input event streams are processed to detect patterns. Such CEP systems may comprise of one or more operators that are represented by a directed acyclic graph. Each operator processes input event streams produced from one or more sources. Event sources might be sensors, upstream operators, other applications, etc. An event (also called primitive event) in the input event streams consists of a meta-data and attribute-value pairs. The event meta-data contains event type, sequence number and timestamp. The event type is used to distinguish events from different types. The event type could be, for example, a stock symbol in a stock market application, a player in a soccer application, etc. Events in the input event streams have global order, e.g., by using the sequence number or the timestamp and a tie-breaker. The attribute-value pairs contain the actual data, e.g., stock quote or player position.

In this work, we focus on a CEP system consisting of a single CEP operator. We assume a window-based CEP system where the input event streams are partitioned into windows (for example, by a window operator) using predicates. The predicates to open and close windows may depend on time (called time-based window), on the number of events (called count-based window), on logical predicates (called pattern-based window) or, on a combination of them [4, 18]. The windows of primitive events are first pushed to the input queue of an operator. An operator continuously gets primitive events from the input queue and processes them by the *process* function which performs the actual pattern matching as shown in Figure 1. The output of the pattern matching are the *complex events*. A primitive event might belong to multiple overlapped windows where it is processed *independently* in each window.

To define patterns, the operator uses an event specification language like Tesla [9], Snoop [8], or SASE [31]. These languages contain several CEP operators: sequence, conjunction, negation, etc.

Example: In intra-day stock trading, an operator receives an event stream containing live stock quote changes of stock A and B throughout the trading day. An analyst wants to detect correlations between a change in A and a change in B in a time period of 1 minute. He formulates a query in the Tesla [9] event specification language:

```
         define Influence(Factor)
         from B() and
[Q_E]    A() within 1min from B
         where Factor = B : change / A : change
```

Let us assume that a window w of 1 minute contains the stock events B_4, B_3, A_2, A_1, where X_i represents event type X and i indicates the event position in stream of ordered events. There exist several instance of each event type in w and hence it is not clear which events of type A should be matched with which events of type B. The generated complex events could be any combination of these event instances. To precisely define which event instances should participate in emitting the complex events, the *selection policy* has been introduced [8, 34]. There are four main selection policies: *first*, *last*, *each*, *cumulative*. In the first selection policy, the earliest event instances are chosen for pattern matching. In the above example, the generated complex events using the *first* selection policy could be $cplx_{13} = (A_1, B_3)$ and $cplx_{24} = (A_2, B_4)$. In last selection policy, the latest event instances are chosen for pattern matching. In the above example, the generated complex events using the *last* selection policy could be $cplx_{23} = (A_2, B_3)$ and $cplx_{24} = (A_2, B_4)$.

In the above example, it is also not clear whether it is allowed to reuse the event instances in performing other pattern matching or they should be considered as consumed events, i.e., not reusing them again. The *consumption policy* [8, 34] defines whether the same event instance can be used in several pattern matching. There are mainly two consumption policies: *consumed* and *zero*. In the above example again, let us assume that the selection policy is *last*. Now, using *consumed* consumption policy results in detecting only one complex event $cplx_{23} = (A_2, B_3)$. Whilst using *zero* consumption policy results in detecting two complex events $cplx_{23} = (A_2, B_3)$ and $cplx_{24} = (A_2, B_4)$, where the event A_2 is reused in $cplx_{24}$. For more information about selection and consumption policy, see [8, 34].

Depending on the used selection and consumption policy, the events in specific positions within a window have different probabilities to be part of the detected complex events. For example, if the used selection policy is *first* and consumption policy is *zero*, then the events in the beginning of the windows have higher probabilities to match the pattern.

We assume that the processing logic in the operator is a black-box, where we do not have any knowledge about their internal states. In this work, we assume that the operator reveals detected complex events, which is a standard assumption in any event processing system. In addition, we assume that the event types are known. Our probabilistic model estimates the utility of events regardless of their order within the matching patterns and also regardless of the used selection and consumption policy.

2.1 Quality of Results

In this paper, we represent the quality of results by 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.

In the above example (cf. 2), using the *first* selection policy and *consumed* consumption policy, two complex events can be detected $cplx_{13} = (A_1, B_3)$ and $cplx_{24} = (A_2, B_4)$. If A_2 is dropped from w, only one complex event can be detected $cplx_{13} = (A_1, B_3)$. This results in one false negative since $cplx_{24}$ is not detected. On the other hand, if A_1 is dropped from w, a *new* complex event is detected $cplx_{23} = (A_2, B_3)$ which results in one false positive and two false negatives since $cplx_{13}$ and $cplx_{24}$ are not detected.

2.2 Problem Statement

To maintain a given latency bound (LB) in the face of system overload, load shedding could be used to drop events from the operator’s input queue. But dropping events might degrade the quality of result. To minimize its impact on the quality of result, the load shedder must drop only those events that have low utility values. The utility of primitive events is measured by their influence on the number of false positives (N_{fp}) and false negatives (N_{fn}), i.e., the quality.

More formally, our objective is to minimize degradation in the quality of result, i.e., minimize ($N_{fp} + N_{fn}$), while dropping events such that the given latency bound LB is met.

3 Probabilistic Load Shedding

In order to minimize the degradation in quality of results, our main idea is to avoid dropping primitive events that could contribute to producing complex events. The question is–how do we identify the importance/utility, in this context, of these primitive events prior to processing them? In real-world applications, event streams have properties that can be exploited to derive the aforementioned importance/utility of a primitive event w.r.t. its probability of contributing to a complex event. An observation is that there is a correlation between **type** and **relative position** within windows of primitive events that constitute complex events. For example, in a soccer game, a sport analyst might be interested in finding a complex event called *man- marking*, i.e., certain defender(s) who always defend against a particular striker. In this case, the ball possession by a striker (*possession-event*) and the defender (*defending-event*) are event types. These two event types have a correlation with each other. Whenever a striker possesses the ball, a defender(s) defends against him in a certain time interval (i.e., relative position in the window), thus producing a complex event. Clearly, in this scenario, position of the primitive events constituting the complex events, i.e., position of *defending-events* relative to the *possession-event*, are correlated. Such correlations also exist in stock market applications. For example, a stock of type IBM may impact a stock of another company within a certain time interval (i.e., relative position in the window), thus resulting in a complex event which detects such an influence. Again, in a different domain, the sensor data set provided by the Intel Research Berkeley Lab shows positive correlation between events of type temperature and events of type humidity [6]. This implies that within a certain interval an increase/decrease in temperature results in an increase/decrease in humidity.

We exploit this correlation, captured by the type and relative position in the window of primitive events, to predict the probability of primitive events to be part of a complex event. In particular, we derive utilities of primitive events in a **window** based on the event **types** and their **relative positions** within the window and use this information to build a probabilistic model that estimates the utility of incoming events in windows. Our load shedder drops only the incoming events that have low utility values within each window, thus minimizing the number of false positives and false negatives.

Next, we explain our probabilistic load shedding strategy in detail. First, we show the architecture of eSPICE. Then, we formally define the utility of primitive events within windows. This is followed by a detailed explanation of our probabilistic learning strategy, how to detect the overload on the system, and how to compute the amount of load to be dropped in order to meet the given latency bound. Finally, we explain the functionality of the load shedder.

3.1 The eSPICE Architecture

To enable load shedding, we extend the architecture of a CEP operator by adding the following components–overload detector, utility models, and load shedder (LS)–as depicted in Figure 1.

The overload detector detects if there exists an overload on the operator. It checks the input event queue size periodically where the incoming windows of primitive events are queued. In case of an overload, the LS drops primitive events from windows to prevent the violation of the defined latency bound (LB). The utility models contain the utility of primitive events in a window and other information that is needed by the LS.

Now, we explain how these 3 components are related. Upon detecting an overload, the overload detector commands the LS to drop events. On receiving this command, the LS uses the utility of primitive events in a window, available from the utility models, to decide on which events to drop.

Please note that load shedding is a time-critical task where it directly affects the CEP system performance and hence it must be lightweight and efficient. As we will see later, our load shedder has very low overhead. Contrarily, building the utility models can afford to be computationally heavy as it is not a time-critical task.

3.2 Utility Models and Their Applications

In this section, we explain, in detail, the utility models and the way they can be used to drop events.

Utility Prediction Function. The utility of a primitive event in a window is defined by its impact on the quality of results. As

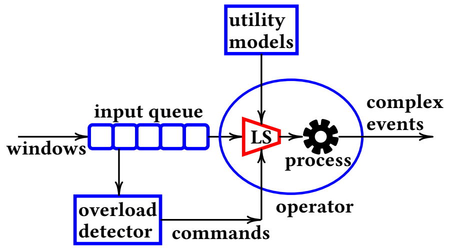


Figure 1. The eSPICE Architecture.

mentioned earlier, we represent utility as the probability of the primitive event to be part of a complex event. Clearly, dropping primitive events that have a high probability to be part of complex events degrades the quality of results. Hence, we avoid dropping these primitive events by assigning high utility values to them. Please recall that we identified the **type** and **position** of an event within a window to be an indicator of whether or not this event has a high probability to contribute to detect a complex event. This implies that the type and position of an event determine its utility.

As a result, to map type and position of events to a utility value, we introduce the utility prediction function $U(T, P)$ that predicts the utility of a primitive event of type T in the position P within a window. As we will see later, this prediction function can be simply implemented based on statistical data collected from the operator.

Utility Threshold and Occurrences. Upon receiving the drop command to drop x events from each window, the LS must find those x events that have the lowest utility values in a window. One simple approach is to *sort* the utility values using an efficient sort algorithm. For example, heap sort has a time-complexity of $O(ws.log_2 x)$, where ws is the number of events in a window [26]. However, this approach requires that the entire window is available to the LS before sorting of the utilities and consequently shedding of events are performed. But waiting until the arrival of all events of a window might introduce a high latency on event processing or might even cause violation of LB. Moreover, sorting needs to be performed in every window which might add additional overhead on the system that already suffers from an overload.

A better approach to avoid the above induced latency and overhead is to find a utility threshold (denoted by u_{th}) that can be used on-the-fly to drop the desired number of events in a given window. In particular, we need a function that maps the number of events to drop per window (x) to a utility threshold u_{th}, i.e., f(x) $\rightarrow u_{th}$. To find the utility threshold u_{th}, we could predict the number of x event occurrences in a window, whose utility is less or equal to the utility threshold u_{th}.

More specifically, in a window w, we define the number of event occurrences, whose utility is less or equal to a certain utility value u as follows: $O(u) = |\{e : U(T, P) \leq u\}|$, where T is the type of event e and P is its position in w. The number of event occurrences $O(u)$ in a window w, as defined above, implicitly represents the cumulative occurrences of those utilities in w, whose values are less or equal to the utility value u and hence, as a short hand, we call $O(u)$ as cumulative utility occurrences.

The utility threshold u_{th} can be calculated using the inverse function of the cumulative utility occurrences $O(u_{th})$, where, given the number of events that should be dropped from each window, we can get the required utility threshold.

Applying Utility Models in Load Shedding. Now, we describe how load shedding is performed in eSPICE. To drop x events from each incoming window, the LS first searches for the cumulative utility occurrences $O(u)$, which has a value $O(u) \geq x$. Then, the LS uses the utility value u as a utility threshold u_{th} to drop those x events from each window.

To use the utility threshold u_{th} and drop events, first, the LS gets the next event e from the input event queue of the operator. Then, for each window w to which the event e belongs, the LS computes the utility value u of the event e in w using the utility prediction function $U(T, P)$. If the event utility u in the window w is greater than the utility threshold u_{th}, the LS keeps the event e in w. Otherwise, it drops the event e from w. The utility threshold u_{th} enables the LS to drop x events from each window.

3.3 Model Building

Having discussed the role of the utility prediction and the threshold prediction functions, in this section, we discuss, in detail, the manner in which we implement these functions. For a clear explanation, let us introduce the following simple running example. We use a pattern matching query that considers a window of 5 events (i.e., window size = 5) and an input event stream consisting of only *two* event types A and B (cf. Figure 3).

Building the Utility Prediction Function. As mentioned in Section 3.2, the utility of a primitive event $U(T, P)$ is represented by the probability of the event to be part of the detected complex events. To predict the utility of primitive events in a window, we collect statistics, from the already detected complex events, on the types and relative positions within windows of *primitive events* that contributed to those detected complex events. More specifically, we count the number of occurrences of each event type in each position in a window that contributed to the detected complex events. The number of occurrences of primitive events within detected complex events provides an incite into the importance (or utility) of the primitive event types and their relative positions within a window. As a result, we simply normalize those number of occurrences to generate the utility (i.e., $U(T, P)$) of a certain event type T in a certain window position P.

These utility values are stored in a table called utility table (denoted by UT). The utility table has (MxN) dimensions, where M represents the number of different event types and N represents the window size ws. Each of its cells $UT(T, P)$ represents the utility of a specific *event type* T in a certain *position* P in a window, where the utility value $U(T, P)$ of an event is stored in $UT(T, P)$.

The values in UT could be too fine-grained which can cause more memory and computational overhead. To avoid this overhead, we limit the number of different utility values by multiplying each cell value in UT by 100 and rounding it to an integer, i.e., $UT(T, P) \in [0, 100]$. Referring to our above example, Table 1 shows a utility table that is generated from the collected statistical data.

Building Utility Threshold and Occurrences. As we discussed in Section 3.2, to drop x events from each window, we should find a utility threshold u_{th} that results in dropping x events from each window, where the utility threshold u_{th} is the inverse function of the cumulative utility occurrences $O(u_{th})$. In particular, we should find a utility value u that is greater or equal to the utility value of x events in a window, i.e., $O(u) \geq x$. Then, we use u as a utility threshold u_{th} to drop x events from each window.

event type \ position	1	2	3	4	5
A	70	15	10	5	0
B	0	60	30	10	0

Table 1. UT generated from the collected statistical data.

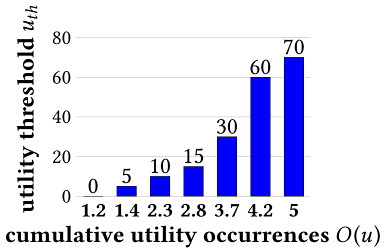


Figure 2. CDT computed from Table 1 (UT) and the predicted position shares in a widnow.

Figure 3. Simple running example.

To find the utility threshold u_{th}, we need to calculate the cumulative utility occurrences $O(u)$ in a window. Since the utilities of events of all types and in all positions in a window are stored in UT, we can determine the cumulative utility occurrences $O(u)$ from UT. The cumulative utility occurrences depend on the distribution of utilities within windows captured in UT.

To predict the utility threshold u_{th}, let us, for now, assume that there is only **one event type** in the input event stream (i.e., $M = 1$) and hence the dimensions of UT become $(1xN)$, recall N is the number of positions (i.e., events) in a window. Since the utility values in UT are between 0 and 100 (recall that $UT(T, P) \in [0, 100]$), there will be a maximum of 101 different utility values, where each utility value in UT may repeat several times. To build the cumulative utility occurrences $O(u)$ for each individual utility value $u \in [0, 100]$, we first count the number of occurrences o_u of each individual utility value $u \in UT$. Once we have the occurrences of each utility value, we can calculate the cumulative utility occurrences. To this end, the number of occurrences o_u of the individual utility values u are accumulated together in a cumulative distribution fashion as follows:

$$O(u) = \begin{cases} o_u, & \text{if } u = 0 \\ o_u + O(u-1), & \text{otherwise} \end{cases} \tag{1}$$

So far, we have assumed that there exists a **single event type** in the input event stream. However, there may be **multiple event types** in the input event stream. In this case, for *each single position* in UT, there exists a utility value for each event type. For example, in Table 1, each single position in UT has two utility values, one utility value for the event type A and one for the event type B. In the table, $UT(A, 1) = 70$ and $UT(B, 1) = 0$. This means that a *single position* in UT is incrementing the occurrences of multiple utilities. As a result, to count the number of utility occurrences o_u, we need to consider each position in UT as a shared position between all event types. More specifically, for each event type, we count a utility occurrence o_u in a certain position in UT as a *fractional value* instead of counting it as a *full occurrence*. We call these fractional values as *position shares in a window*. We could predict the position shares in a window between different event types from the distribution probability of the primitive events within the window. The position shares in a window $S(T, P)$ of a primitive event of type T at position P in the window equals the probability of this event type T to come at the position P in the window.

Now, to compute the cumulative utility occurrences $O(u)$ in case of **many event types**, we count the occurrences o_u of the utility value u in UT as a fractional value by its corresponding values from the position shares in a window. For each utility value $u = UT(T, P)$ for the event type T at position P in UT, we increase the number of occurrences o_u by $S(T, P)$. The cumulative utility occurrences $O(u)$ is then computed as in case of a single event type using equation 1.

We store the cumulative utility occurrences $O(u)$ in an array called CDT, where the utility values u are used as indices and the cumulative utility occurrences $O(u)$ are used as the actual values, i.e., $CDT(u) = O(u)$. CDT is a single dimensional array of size (101), which is the maximum number of different utility values in UT. An *index* u in CDT represents a utility value u in UT and its cell value $CDT(u)$ represents the cumulative utility occurrences $O(u)$ of the utility value u.

Since the utility threshold u_{th} is the inverse function of the cumulative utility occurrences $O(u)$, we extract u_{th} from CDT. To find a utility threshold u_{th} which drops x events from each window, we iterate over CDT to find a cell value $CDT(u)$ that is $\geq x$, which means that the number of primitive events with utility values less or equal to u occur at least x times in each window. Hence, using u as a utility threshold drops at least x events from each window. We explain utility threshold prediction further with the help of our running example. Figure 2 shows the CDT computed from UT in Table 1 and the predicted position shares in a window. Now, to drop $x = 2$ events from each window, in the figure, $CDT(10) = 2.3 > x$. Thus, to drop $x = 2$ events from each window, we use the utility threshold $u_{th} = 10$.

Algorithm 1 explains the construction of CDT from both UT and the predicted position shares in a window. The algorithm first counts the number of occurrences o_u of each individual utility value u in UT (cf. lines (2-5)). It iterates over each cell in UT (cf. lines (2-3)) and gets its value $u = UT(T, P)$, i.e., the utility of the event type T at the position P in the window (cf. line 4). Then, in line 5, the algorithm increments the cell value in a temporary array *temp* which is at index u by the fractional value $S(T, P)$. Since the utility values are used as indices in CDT, they are already sorted in an ascending order. Finally, the algorithm accumulates the values in CDT starting from index 0 where $CDT(u) = CDT(u) + CDT(u - 1),\ u = 1..100$ (cf. lines(7-9)).

Algorithm 1 Building CDT table.

```
1:  computeCDT () begin
2:   for $T = 0$ to $(M-1)$ do        ▷ $M$: Number of different event types
3:       for $P = 0$ to $(N-1)$ do        ▷ $N$: the window size $ws$
4:           $u = UT(T, P)$
5:           $temp(u)$ += $S(T, P)$        ▷ $temp(u) = o_u$
6:   //accumulate utility values in an ascending order.
7:   $CDT(0) = temp(0)$
8:   for $u = 1$ to 100 do
9:       $CDT(u) = temp(u) + CDT(u-1)$
10: end function
```

Please note that building UT and CDT is a continuous task where they are periodically updated .

3.4 Overload Detection

Having explained the way utility models are built, we now provide details on when the LS should drop events and how many and in which interval should events be dropped.

To detect an overload on an operator, the overload detector periodically monitors the input event queue and calculates the estimated latency for the incoming events (denoted by $l(e)$). It compares $l(e)$ with the defined latency bound LB and decides to drop events if LB might be violated.

The value of $l(e)$ depends on event processing latency (denoted by $l(p)$) and event queuing latency (denoted by $l(q)$), in fact, $l(e) = l(q) + l(p)$. Event processing latency $l(p)$ represents the time an event needs to be processed by the operator in all windows. $l(p)$ is calculated from the throughput of the operator (denoted by th). The throughput th represents the maximum number of events the operator can process per second. Event queuing latency $l(q)$ represents the time an event must wait before it gets processed by the operator. This time depends on the number of queued events n before this event e in the input event queue and on $l(p)$, i.e., $l(q) = n * l(p)$. This means that an event e in position n in the input event queue has an estimated latency $l(e) = (n-1) * l(p) + l(p) = n * l(p)$.

From the given latency bound LB and the event processing latency $l(p)$, we can get the maximum allowed queue size (denoted by q_{max}) before violating LB, where $q_{max} = LB/l(p)$.

Waiting until the queue size (denoted by q_{size}) equals q_{max} to start dropping events might be too late and can cause LB violation. Therefore, we start dropping events, if the following inequality holds: $q_{size} > f.q_{max}$, where $f \in [0,1]$, see Figure 4. A high f value, on one hand, avoids unnecessarily dropping events– in cases the events are only queued for a short time as in short burst situations. But on the other hand, it might force the LS to drop events with high utility values to avoid LB violation– in case the queue size gets close to q_{max}. Later, we explain how to choose an appropriate f value.

Dropping Interval. So far, we have considered dropping x events per window. However, the window size might not be the best dropping interval to meet the given latency bound LB. The reason is that as the LS starts dropping events when $q_{size} > f.q_{max}$, the buffer that we have before violating the latency bound (LB) is of size ($q_{max} - f.q_{max}$) events (cf. Figure 4). More specifically, we need to drop x events from at least every ($q_{max} - f.q_{max}$) events (i.e., dropping interval) in order to meet LB. Therefore, please note that the dropping interval must be less or equal to ($q_{max} - f.q_{max}$).

As a result, if the window size ws is less or equal to this buffer size (i.e., $q_{max} - f.q_{max}$), then the interval of dropping x events is preserved and the utility threshold u_{th} can be calculated for the entire window. However, if the window size ws is greater than the buffer size, there is a risk of LB violation, especially if the utility values are not evenly distributed in windows, e.g., all events with high utilities come together in a certain region of the window. In this case, the utility threshold u_{th} will result in dropping x events from each window but not necessarily from each dropping interval (i.e., the buffer size) if the size of the high utility region of the windows is greater than the buffer size. This might result in LB violation.

Therefore, we must partition the window into smaller partitions of size less or equal to the buffer size, i.e., $q_{max} - f.q_{max}$ (as can be seen in Figure 4) and drop x events from each partition. While the partition size cannot be greater than the buffer size (cf. the above mentioned constraint), of course, it can be less than the buffer size. However, larger the partition size is, greater is the probability to find low utility values to drop, resulting in better quality. As a result, we try to use a partition size which is as large as possible (of

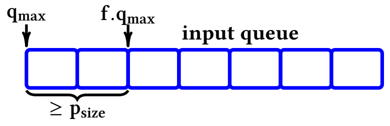


Figure 4. Partition Size.

course, the upper bound being the buffer size). More specifically, we partition a window in ρ partitions of equal sizes, where $\rho = ceil(\frac{ws}{q_{max}-f.q_{max}})$. As a result, the partition size $p_{size} = \frac{ws}{\rho}$. We use the partition size as a *dropping interval* in which x events should be dropped. Therefore, we cannot use the utility threshold u_{th} that comes from a full window but instead we have to use a utility threshold u_{th} for each partition in order to drop x events in each dropping interval.

We already discussed how to compute CDT, i.e., the cumulative utility occurrences $O(u)$, for a complete window. However, since a window might be divided into more than one partition (when $\rho > 1$), we must compute for each partition its own CDT. Please note that UT will be calculated as before. However, the utility threshold u_{th} needs to be calculated based on the partition size p_{size} within which shedding must be performed. Therefore, we compute CDT for each partition of size p_{size} within UT. So, now, to drop x events from each partition of the incoming windows, each partition has its own utility threshold u_{th}.

Dropping Amount. The dropping amount represents the number of primitive events that must be dropped from each partition of each window. Determining how many events x to drop per partition depends on the input event rate and the operator throughput th. The overload detector computes the difference δ between the input event rate R and th, where $\delta = R - th$, to get the extra incoming events per *second*. Then, the number of events x to drop per partition is computed as follows: $x = \delta.\frac{p_{size}}{R}$, where $\frac{p_{size}}{R}$ represents the partition size in seconds.

Appropriate f Value. As we mentioned above, using a high f value prevents dropping events in short burst cases and hence decreases the degradation in the quality of results. However, the f value controls the partition size p_{size}, and hence using a high f value forces us to use a small partition size to avoid LB violation. A small partition size might result in dropping events that have high utility values. That can happen if all events in a partition have high utility values. Therefore, we should choose a minimum f value that still allows to have a partition size which avoids dropping high utility events.

Fortunately, we already have the distribution of utilities within a window captured in UT. We can take advantage of this knowledge to determine f value. To find the f value, we propose to cluster the utilities in UT into several classes of importance. The goal is to partition the windows depending on the f value into one or more partitions, where in each partition, there exists at least x events from the low utility classes. This way, in each partition, the low utility events can be dropped, hence reducing the degradation in the quality of results. Therefore, we can choose the f value that ensures the above partition size.

3.5 Load shedding

Now, we explain, in detail, the functionality of eSPICE's load shedder (LS) component. Events are dropped from individual windows without affecting other windows. An event might be dropped from one window while it is still there in other windows as the event utility $U(T, P)$ can be different in different windows, since the event position P is different in different windows. The LS checks for each incoming event in a window and decides on whether or not to drop it depending on the event utility in UT and on the utility threshold u_{th} of the partition to which the event belongs. Hence, the LS must be lightweight since it is performed for every event in a window.

Upon receiving the drop command from the overload detector, the LS searches for the utility thresholds corresponding to each partition of an incoming window. Note that the entire window can also be a single partition, i.e., there is only one partition ($\rho = 1$), if $ws \leq (q_{max} - f.q_{max})$ (cf. previous section). Having noted the utility thresholds for every partition of a window, the LS proceeds to drop events from the incoming windows. So, for each event e in a window, the LS gets its utility $U(T, P)$ from UT and also determines the partition ($part$) in a window to which the event e belongs, both in O(1) time-complexity. Then, the LS compares the event utility $U(T, P)$ with the utility threshold u_{th} of the corresponding partition ($part$) to decide on whether or not to drop e from the window. If the utility of the event is less or equal to the utility threshold of its corresponding partition, the LS drops the event.

Algorithm 2 explains the LS functionality more formally. If $q_{size} > f.q_{max}$, the overload detector requires the LS to activate the shedding. It also sends drop commands which contain the number of events x to drop per partition to LS. The LS receives drop commands from the overload detector where it firstly calculates the utility threshold u_{th} for each partition depending on the required number of events x to drop per partition (cf. lines 1-7). To calculate the utility threshold $u_{th}(part)$ for each partition $part$, the LS iterates (cf. lines 2-3) over its corresponding CDT to search for a value $CDT(part, u)$ which is $\geq x$ (cf. line 4). Then, the index u of this value $CDT(part, u)$ is used as the utility threshold $u_{th}(part)$ for this partition $part$ (cf. line 5).

In case load shedding is active, for each event e in the incoming windows, the LS checks if it needs to drop the event e (cf. lines 8-17). First, the LS finds the partition in the window to which the event e belongs (cf. line 12). Then, the LS checks if the utility value $UT(T, P)$ of this event e in UT is less or equal to the utility threshold $u_{th}(part)$ of its calculated partition (cf. lines 13-16)—just a simple lookup in UT. It then returns true if the event should be dropped, otherwise false. This shows that our load shedder is extremely lightweight and it takes the shedding decision in O (1) time-complexity for each event in a window.

Algorithm 2 Load shedder.

```
1:  getUtilityThresholdForEachPartition ($x$) begin
2:    for $part = 0$ to $(\rho - 1)$ do
3:        for $u = 0$ to 100 do
4:            if $CDT(part, u) \geq x$ then
5:                $u_{th}(part) = u$
6:                break    ▷ break the inner loop and proceed to the next partition.
7:  end function
8:  applyLS ($T$, $P$) begin    ▷ $T$: event type & $P$: event position in the window
9:    if $!LS.active$ then
10:       return $false$
11:   else
12:       $part = \frac{n}{p_{size}}$
13:       if $UT(T, P) \leq u_{th}(part)$ then
14:           return $true$
15:       else
16:           return $false$
17: end function
```

3.6 Extensions

In this section, we explain three extensions to our load shedding approach—handling variable window size, using bins for large windows, and retraining the model. Handling variable window size enables our approach to work with windows of different sizes. While, the use of bins optimizes our approach and enables it to work with large windows.

Handling Variable Window Size. The incoming windows might have a variable window size ws depending on the window splitting strategies. As mentioned above, in CEP systems, there exist three main window splitting strategies—count-based, time-based and pattern-based. In count-based, ws is always fixed while in time-based and pattern-based, ws might change depending on the input event rate or content of the events [18].

As explained earlier, in order to implement the utility prediction function, we use UT which has a fixed number of event positions N, where $N = ws$. However, if the window size ws varies and is not fixed, we need a way to find N. Therefore, to handle variable window size, we profile the operator and choose N as the average seen window size. Since N might be different from the incoming window sizes, in the following, we explain the required modifications to our approach during both model building and load shedding to incorporate variable window size.

During Model Building: We need a way to map the event positions in windows to the event positions in UT that has a fixed number of positions N. To do that, we normalize the size of incoming windows to N. For each incoming window, if $ws > N$, we scale down w where more than one position in w is mapped to a single position in UT. On the other hand, If $ws < N$, we scale up w where each position in w is mapped to one or more positions in UT. The scaling factor sf can easily be computed as follows: $sf = \frac{ws}{N}$. For example, let $N = 100$ and $ws = 200$, then $sf = \frac{200}{100} = 2$. This means that every two positions in w is mapped to a single position in UT.

During Load Shedding: The window size may also vary while performing load shedding. So, in this case while processing every incoming event e of a window w, the LS must determine the relative position of the event e within the window w, instead of the exact position. In this way, the LS can map the learned utility values in UT to the event e. To map relative position of the event e in the window w to the exact position in UT, we again scale down ws if $ws > N$ and scale up ws if $ws < N$. Since during scaling up ws, an event e in w is mapped to more than one cell in UT, the utility of e is the average value of all these cell values in UT.

As mentioned above, the cumulative utility occurrences $O(u)$, which are stored in CDT, are computed from UT that has a fixed number of positions N. In case of varying window sizes, the utility threshold u_{th} is calculated from CDT without any modification. This is because, the utility values in UT already capture the variation in the window size. So, the calculated utility threshold u_{th} from CDT implicitly scales up/down depending on the window size.

The problem with variable window size during load shedding is that we process events on their arrival without waiting until the end of the windows. Thus, the incoming window size is not known at the time when the LS performs lookup in UT to get the utility of an event in a window based on its relative position. It is not possible to get a relative position if the incoming window size is not known. However, the window size is important for the lookup and we must predict it. For example, in case of time-based window, the event input rate could be used to predict the window size. Predicting the window size is already researched in literature [18] and will not be the focus of this paper.

Using Bins for a Large Window Size. The average seen window size N might be too large. This might result in a huge size of UT, thus wasting computing resources. Therefore, bins of size bs are used to map several neighboring positions for each specific event type in a window to one single position in UT, thus reducing its size. In Section 4, we discuss more about the impact of the bin size on the quality of results.

Model Retraining. If the distribution of the input event stream changes over time, the constructed model becomes inaccurate, which may adversely impact the quality of results. Therefore, in this case, we must retrain the model to capture the changes in the input event stream. We can either periodically retrain the model to capture these changes or we can use a statistical approach that triggers the need to retrain the model (we leave this approach for future work).

4 Performance Evaluations

Next, we evaluate the performance of eSPICE by analyzing its impact on the quality of results when the input event rate exceeds the operator throughput th.

4.1 Experimental Setup

Here, we describe the evaluation platform, the baseline implementation, datasets and queries used in the evaluations.

Evaluation Platform. We run our evaluation on a machine which 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 the operator in a single thread which is used as a resource limitation. Please note, that the performance of eSPICE is independent of the parallelism degree of the operator. We implemented eSPICE by extending a prototype CEP framework which is implemented using Java.

Baseline. As evaluation results showed that a completely random event shedder is comprehensively outperformed by eSPICE, we did not think it would be a fair comparison and therefore looked at comparing eSPICE with a baseline that is similar to state-of-the-art solutions. We implemented a load shedding strategy which is similar to the strategy in [12] as a baseline strategy (denoted by BL). Our implemented baseline strategy also captures the notion of weighted sampling techniques in stream processing [29]. BL assigns utility values to the primitive events in a window depending on the repetition of those primitive events in the pattern and on their frequencies in windows. An event type receives a higher utility proportional to its repetition in a pattern. Depending on the event utility, BL decides the amount of events that should be dropped from each event type in a window, where it uses uniform sampling to drop those required amounts from each event type. BL, as in [12], does not consider the order of events in a pattern and in input event streams.

Datasets. We employ two different datasets from real-world. First, we use a stock quote stream from the New York Stock Exchange (NYSE). This dataset contains real intra-day quotes of 500 different stocks from NYSE collected over two months from Google Finance [2]. The quotes have a resolution of 1 quote per minute for each stock symbol. We refer to this dataset as the *NYSE Stock Quotes* dataset. Second, we use a position data stream from a real-time locating system (RTLS) in a soccer game [19]. Players, balls and referees (called objects) are equipped with sensors that generate events which contain their position, velocity, etc. The sensor data are generated at a high rate causing high redundancy. Thus, we filter redundant events and keep only one event per second for each object. We refer to this dataset as the *RTLS* dataset.

Queries. We employ four queries (Q1, Q2, Q3, Q4) that cover an important set of operators in CEP: sequence operator, sequence with any operator, and sequence operator with repetition, all with skip-till-next/any-match [4, 8, 9, 11, 31, 33]. Moreover, the queries use both **time-based** and **count-based** sliding window strategies with **different predicates**. Furthermore, to study the robustness of our load shedding with different *selection policies*, we implemented all queries with the *first* and *last* selection policies. All queries skip the intermediate not matching primitive events, i.e., skip-till-next/any-match. The queries are as follows :

Q1 (sequence with any operator): uses the RTLS dataset. It detects a complex event when any n defenders of a team (defined as DF) defend against a striker (defined as STR) 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. We use two players as strikers; one striker from each team. Q1 is of form: seq (STR; any (n, DF_1, DF_2, .., DF_n)), where DF_x is the defend event of the player x.

Q2 (sequence with any operator): is adopted from related work [17]. It detects a complex event when any n rising or any n falling stock quotes of any stock symbol (defined as RE or FE, respectively) are detected within ws seconds from a rising or falling quote of a leading stock symbol (defined as MLE). The leading stock symbols are composed of a list of 5 technology blue chip companies. Q2 is of form: seq (MLE; any (n, RE_1, RE_2, ..,RE_n)) or seq (MLE; any (n, FE_1, FE_2,..,FE_n)), where RE_x or FE_x is rising or falling event of the stock company x.

Q3 (sequence operator): detects a complex event when rising or falling stock quotes of 20 certain stock symbols (defined as RE or FE, respectively) are detected within ws events in a certain sequence. Q3 is of form: seq (RE_1; RE_2;..;RE_{20}) or seq (FE_1; FE_2;..;FE_{20}), where RE_x and FE_x are defined as in Q2.

Q4 (sequence operator with repetition): detects a complex event when 10 rising or 10 falling stock quotes of certain stock symbols (defined as RE or FE, respectively) *with repetition* are detected within ws events in a certain sequence. Q4 is of form: seq (RE_1; RE_1; RE_2; RE_3; RE_2; RE_4; RE_2; RE_5; RE_6; RE_7; RE_2; RE_8; RE_9; RE_{10}), where RE_x is defined as in Q2. The sequence for falling quotes is similar.

4.2 Experimental Results

In this section, first, we evaluate the impact of our probabilistic LS strategy (eSPICE) on the quality of results, particularly the number of false positives and false negatives, and compare its results with the results of BL. Then, we show the impact of variable window size and bin size on the quality of results and analyze the overhead of the LS.

If not noted otherwise, we employ the following settings. The number of complex events per window is one. For Q1 and Q2, we use a *time-based* sliding window. In both queries, a *logical* predicate is used to open new windows. For Q1, a new window is opened for each incoming striker event (STR). While for Q2, a new window is opened for each incoming event of the leading stock symbols (MLE). Q3 and Q4 use *count-based* sliding window. In Q3, we also use a *logical* predicate where a new window is opened for each incoming event of the leading stock symbols (MLE). For Q4, a *count-based* predicate is used where a new window is opened every 100 events, i.e., the slide size equals 100 events. We use a latency bound $LB = 1$ second and an f value $= 0.8$. Moreover, we stream the datasets from stored files to the system with an event input rate that is less or equal to the maximum processing rate of the operator (i.e., the operator throughput (th)) until the model is built. After that, we increase the input event rate to a rate that is higher than the operator throughput th by 20% (denoted by $R1$) and 40% (denoted by $R2$). We execute several runs for each experiment and show the mean value and standard deviation.

Impact on the quality of results and the given latency bound. First, we show the impact of eSPICE on the given latency bound and on the quality of results, i.e., the number of false negatives and false positives. The quality is influenced by the input event rate and the ratio of pattern size to window size. Therefore, we show results using different input event rates and different ratios of pattern size to window size.

To evaluate the performance of eSPICE, we run experiments with Q1, Q2, Q3 and Q4. For Q1 and Q2, we use a variable pattern size with a fixed window size to get different ratios of pattern size to window size. For Q1, we use a window size ws of 15 seconds (≈ 700 events) and the following pattern sizes (i.e., number of defenders): n= 2, 3, 4, 5, 6. The window size ws for Q2 is 240 seconds (≈ 2000 events) and the pattern is of the following sizes: n= 10, 20, 30, 40, 50, 60, 70, 80. Since the pattern size is fixed in Q3 and Q4, we use a variable window size to get different ratios of pattern size to window size. For both Q3 and Q4, we use the following window sizes: ws =300, 600, 1200, 1500, 1800, 2000 events.

Number of false negatives. Figures 5a and 5b show results for Q1 with the first and last selection policy, respectively, while Figures 5c and 5d show results for Q2 with the two selection policies. Similarly, Figures 5e and 5f show results for Q3 and Q4 using the first selection policy. The x-axis in Figures 5a, 5b, 5c and 5d represents the pattern size, while in Figures 5e and 5f, the x-axis represents the window size. In all figures, the y-axis represents the percentage of false negatives. Moreover, all figures show the result of applying eSPICE and BL while using the two input event rates $R1$ and $R2$.

Using a large ratio of pattern size to window size may result in making the pattern more sensitive to any event drop. Thus, dropping even a few events may hinder detecting the pattern and hence it may result in more false negatives depending on the query. This can be observed in the following results. In Figure 5a, using the input event rate $R1$ and first selection policy, the percentage of false negative as a result of using eSPICE (cf. Figure 5a , the blue line) is 9% when the pattern size is 2 and it increases with the increment in the pattern size, where it reaches 21.2% with a pattern of size 6. Similarly, the percentage of false negatives using BL increases from 45.6% to 55.9% with the pattern sizes 2 and 6 (cf. Figure 5a , the red line). Also, the percentage of false negatives increases with the higher input event rate $R2$ for both eSPICE and BL (cf. Figure 5a , the green and brown lines, respectively). This is because higher input rate implies more dropping. However, it shows a similar behavior with the increase in the pattern size as in $R1$. In Figure 5a, eSPICE produces better results than BL in all cases where the performance of eSPICE is higher than the performance of BL by up to 5 and 3.2 times using the event input rates $R1$ and $R2$, respectively.

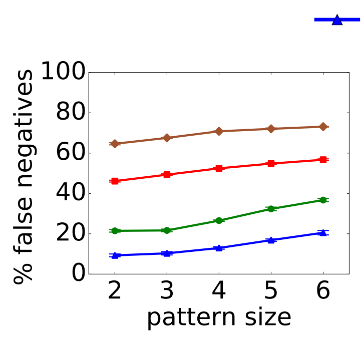
Q1, using last selection policy, shows similar behavior to the first selection policy behavior as depicted in Figure 5b. The percentage of false negatives for eSPICE and the input event rate $R1$ increases from 4% to 6% for increasing pattern sizes of 2 to 6. Meanwhile, the percentage of false negatives for BL increases from 27% to 42% for the same pattern sizes of 2 to 6 and with the same input rate $R1$. This shows that eSPICE performs better than BL by up to 7 times. Similar results are shown when using the input event rate $R2$.

The results of Q2 has similar behavior to the results of Q1. Figure 5c shows that eSPICE performs better than BL by up to 30 and 24 times using the input event rates $R1$ and $R2$, respectively, while using the first selection policy. The performance of eSPICE while using last selection policy is very similar to that of the first selection policy, where it significantly outperforms BL for both input event rates $R1$ and $R2$.

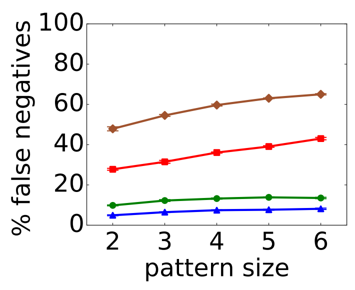
Figures 5e and 5f, i.e., running Q3 and Q4 with the first selection policy, show that the performance of eSPICE is much better using sequence operator that matches an exact pattern, where the percentage of false negatives is almost zero with both input event rates $R1$ and $R2$. Using the *sequence* operator ensures that every time only the *exact same event types* would match the pattern and construct the complex events. This results in higher utility values for those event types and the shedding is extremely accurate. On the other hand, the *sequence with any* operator allows any event type to match the pattern and hence it results in distributing the utility values more sparsely between different available event types in windows. This is the reason why eSPICE performs exceedingly well for sequence operator. As a result, the performance of eSPICE for Q3 and Q4 is enormously better than the performance of BL which produces a high percentage of false negatives. Moreover, our results clearly show that the presence of repetitions in the sequence operator does not impact the performance of eSPICE (cf. 5e and 5f).

Number of false positives. Figures 6a and 6b show results for Q1 and Q3, respectively, with the first selection policy. We do not show Q2, Q4, and the last selection policy as they have similar results. In Figure 6a, the x-axis represents the pattern size while the x-axis in Figure 6b represents the window size. In both Figures, the y-axis represents the percentage of false positives. Moreover, both figures show the results of applying eSPICE and BL while using the two input event rates $R1$ and $R2$.

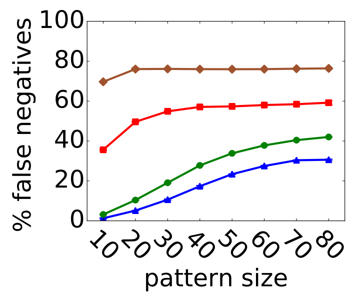
The percentage of false positives in Figure 6a shows a similar behavior to the percentage of false negatives in Figure 5a, i.e., Q1 with the first selection policy. The percentage of false positives increases with the increment in the pattern size, i.e., the increment in ratio of pattern size to window size. This is because, the probability to drop a primitive event that contributes to detect a complex event in a window increases with a higher pattern size, thus increasing the percentage of false negatives as we showed above. Moreover, since



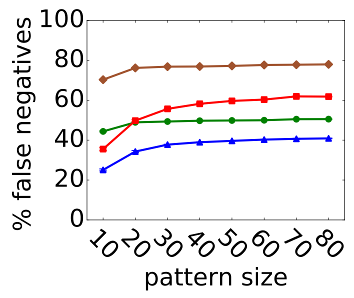
(a) Q1: First selection policy



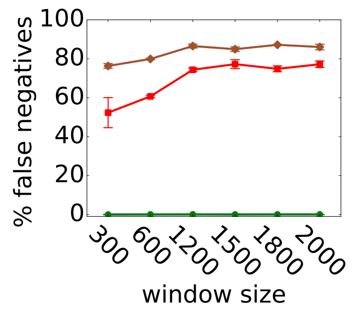
(b) Q1: Last selection policy



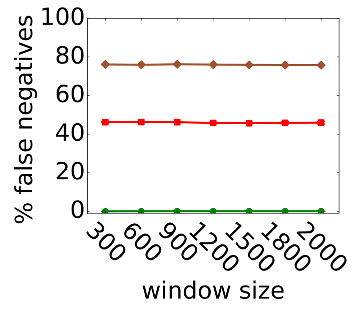
(c) Q2: First selection policy



(d) Q2: Last selection policy

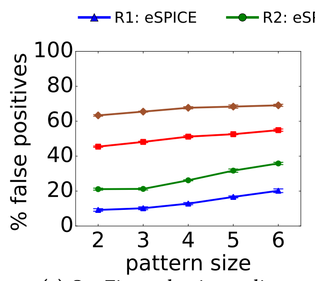


(e) Q3: First selection policy

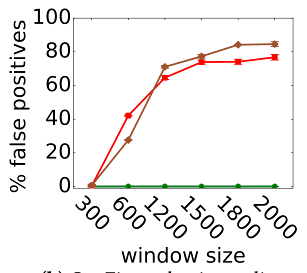


(f) Q4: First selection policy

Figure 5. False negatives for Q1, Q2, Q3 and Q4 with input event rates $R1$ and $R2$.



(a) Q1: First selection policy



(b) Q3: First selection policy

Figure 6. False positives for Q1 and Q3 with $R1$ and $R2$.

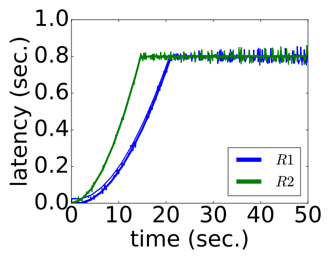


Figure 7. Event processing latency.

Q1 uses the *sequence with any* operator, the probability to find any other primitive event, i.e., any other defender event, alternative to the dropped primitive event is high, which can result falsely in detecting a new complex event, i.e., a false positive. As in false negative experiment, the percentage of false positives increases with the input event rate. eSPICE performs better than BL up to 4.8 and 3.2 times using the event input rates $R1$ and $R2$, respectively.

The percentage of false positives in Figure 6b also shows a similar behavior to the percentage of false negatives in Figure 5e, i.e., Q3 with the first selection policy, where the percentage of false positives is almost zero for eSPICE. However, the percentage of false positives for BL increases with the increment in the window size. This is because, Q3 uses the *sequence* operator which matches an exact pattern. Thus, if a primitive event that contributes to a complex event in a window is dropped, it is hard to find an alternative primitive event to the dropped primitive event in smaller windows. The probability to find this alternative primitive event in a window increases with the increment in the window size, thus increaing the number of falsely detected events, i.e., increasing the percentage of false positives.

Maintaining the given latency bound. The main goal of eSPICE is to maintain the given latency bound. Hence, here, we discuss the ability of eSPICE in keeping the given latency bound. Figure 7 shows the incurred event latency (l_e) in case of running Q1 with the input event rates $R1$ and $R2$. The results of other queries show similar behavior and hence they are not shown. The figure shows that eSPICE never violated the given latency bound ($LB = 1$ second) and it always keeps the event latency around ($f * LB$) which is 800 milliseconds in this experiment.

Impact of variable window size on the quality of results. Now, we show the impact of variable window size on the quality of results. Using time-based or pattern-based sliding window may result in splitting the incoming event stream into windows of different sizes. However, UT has a fixed number of positions/events N, where N represents the average window size, given $bs = 1$. Hence, we must map the incoming windows of different sizes to N as we showed above in Section 3.6.

The ideal window size should be N, however, in case the incoming windows are larger or smaller than N, the quality of results might degrade because of the variations in relative positions of events in windows . To evaluate that, we run experiments with Q1 and Q2 where we use several window sizes during model building to enforce having different number of events per window.

For Q1, we use a pattern size of $n = 5$. Moreover, we use a time-based window of the following sizes: ws= 12 , 14, 16 ,18 and 20 seconds. The average seen window size is ≈ 800 events and hence we use $N = 800$ to build UT. As the window size $ws = 16$ seconds contains around 800 events (≈ N), we use it as a reference window size in our results and refer to it as a window of size 100%. We represent the window sizes as percentage values compared to the reference window size (i.e., $ws = 16$ seconds) and hence the used windows are of the following sizes: 75%, 87%, 100%, 112% and 125%.

For Q2, we use a pattern size of $n = 20$. Again, we use a time-based window of the following sizes: ws= 180 , 200, 240 , 260 and 300 seconds. The average seen window size is ≈ 2000 events and hence we use $N = 2000$ to build UT. The window size $ws = 240$ seconds contains around 2000 events (≈ N) therefore we use it as a reference window size in our results and refer to it as window of size 100%. Again, we represent the window sizes as a percentage value compared to $ws = 240$ seconds and hence the used windows are of the following sizes: 75%, 83%, 100%, 108% and 125%.

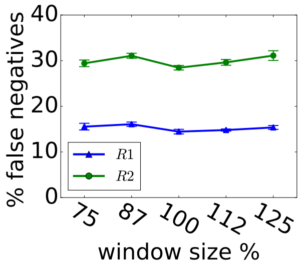
For both queries, during the model building, we change the window size between the above given window sizes randomly to ensure that our model has learned from several window sizes and not only from one window size. During load shedding, we use one of the window sizes of the above given window sizes to check the impact of this window size on the quality of results.

Figure 8 depicts the percentage of false negatives for both Q1 and Q2. The x-axis represents the percentage of window size compared to the reference window size and the y-axis represents the percentage of false negatives. Figure 8a shows results for Q1 with the two event input rates $R1$ and $R2$, respectively while Figure 8b shows results for Q2 with the two rates, all using the first selection policy. We observed similar results for Q1 and Q2 using the last selection policy and also for the number of false positives experiments and hence we don't show them.

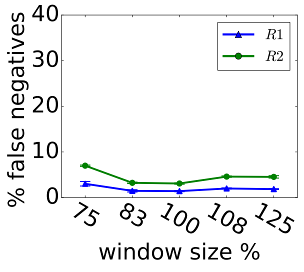
Figure 8a shows that the percentage of false negatives is only slightly influenced by the used window size with both input event rates $R1$ and $R2$. Hence, more than one event in a window can be mapped to a single position in UT in case $ws > N$ or one event in a window can be mapped to several positions in UT when $ws < N$ without having a considerable impact on the number of false negatives.

The impact of window size is clearer in Figure 8b, i.e., in Q2. In this figure, the percentage of false negatives increases when the difference between N and the window size increases. The reason behind this is that Q2 has a longer pattern size than Q1 which makes it more sensitive to the relative event positions in windows. Moreover, the number of event types (i.e., MLE) that start a new match in Q2 is higher than the number of event types that start a new match in Q1 (only two strikers). In Q2, each event in MLE may impact different stock companies and hence the utility values are more distributed between different event types and over the whole window. Meanwhile, in Q1, the distribution of the utilities is more focused on specific number of defenders.

Impact of bin size on the quality of results. A big bin size might degrade the quality of results since it reduces the accuracy in UT of the important positions in the incoming windows. To analyze the impact of bin size on the quality of results, we run experiments with Q1 and Q2. We use a pattern of size $n = 5$ and $n = 20$ and

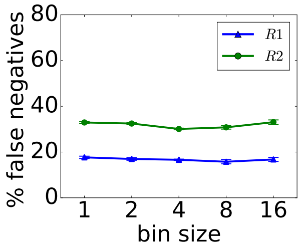


(a) Q1: First selection policy

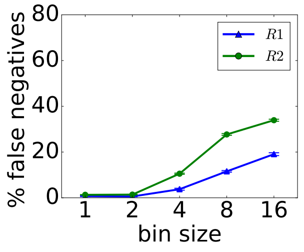


(b) Q2: First selection policy

Figure 8. Impact of variable window size on the quality.



(a) Q1: First selection policy



(b) Q2: First selection policy

Figure 9. Impact of bin size on the quality.

a window of size $ws = 15$ seconds and $ws = 240$ seconds for Q1 and Q2, respectively. In addition, we use the following bin sizes for both queries: $bs = 1, 2, 4, 8, 16, 32, 64$.

Figure 9 depicts the percentage of false negatives for both queries with the first selection policy. The x-axis represents the bin size and the y-axis represents the percentage of false negatives. We observed similar results for Q1 and Q2 using the last selection policy and also for the number of false positives experiments and hence we don't show them.

Figure 9a depicts results for Q1 with the rates $R1$ and $R2$ where it shows that the percentage of false negatives is slightly influenced by the used bin size for both input event rates $R1$ and $R2$.

Figure 9b depicts results for Q2 with the rates $R1$ and $R2$ where it shows that the percentage of false negatives increases with the used bin size. The reason here is again similar to the reason in the variable window size experiment.

Run-time overhead of the LS. Load shedding is used in systems that already face overload and hence the LS overhead must be considerably small compared to the event processing overhead. Our LS performs only a single lookup in the utility table UT to decide whether or not to drop an event from a window and hence its time-complexity is O(1). Thus, it is a lightweight load shedding strategy.

An important parameter that impacts the LS overhead is the window size. A large window may not fit in the system caches and cost higher lookup time and hence higher overhead.

To show the overhead of the LS, we run experiments for Q2 with the two input event rate $R1$ and $R2$ and use a window of the following sizes: $ws = 240, 360, 480, 960, 1920$ seconds, where the approximate window sizes in events are 2000, 3000, 4000, 8000 and 16000 events, respectively. We used these approximate window sizes in events as a dimension for UT, i.e., $N = ws$. We observed similar behavior for other queries and hence we do not show them.

Figure 10 depicts the overhead of the LS for Q2. The x-axis represents the used window size and the y-axis represents the percentage

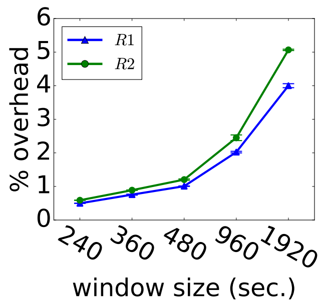


Figure 10. Q2: Overhead of the LS.

time the LS needs compared to the actual event processing time. As expected, the overhead of our LS increases with the used window size. In the figure, the overhead increases from less than 1% with the window of size 240 seconds (≈ 2000 events) to ≈ 5% with the window of size 1960 seconds (≈ 16000 events). Please note that the result shown for the window size 1960 seconds is with respect to a large table size where $M = 500$ (number of stock companies) and $N = 16000$. However, the overhead is still small compared to the actual event processing time. Hence, our load shedding strategy can maintain the given latency bound with a low overhead. Moreover, the overhead of the window size can be reduced by increasing the the bin size (bs). In addition, improving the utility table locality in the memory can further reduce the overhead of LS.

Results Discussion. eSPICE performs much better than BL for all queries, datasets and selection policies. However, the performance of eSPICE varies for different classes of operators. The performance of eSPICE is exceptionally good for the *sequence* and *sequence with repetition* operator with a negligible number of false negatives and positives. The *sequence* operator ensures that every time only the same event types would match the pattern and this results in higher utility values for those event types. On the other hand, the *any* operator matches any event regardless of its type. Hence, the utility of the events are more sparse which adversely impacts the performance of eSPICE. Further, eSPICE shows its robustness against variable window size and bin size, where the quality of results is only slightly influenced by a window size which is different from N or by a higher bin size. Moreover, the overhead of LS component in eSPICE is very low compared to the actual processing overhead which makes eSPICE suitable for real-time complex event processing.

5 Related Work

Complex event processing has emerged as an important paradigm to process event streams on-the-fly. Different event definition languages [8, 9, 31] have specified the rules for CEP systems. To precisely match the events, [8, 9, 34] define selection and consumption policies. CEP systems are used to process huge event streams and required to provide a high performance. Hence, the CEP operator graph is usually distributed on several compute nodes and each operator might be parallelized to speedup its processing [1, 20]. A powerful parallelization technique is data-parallel CEP [1, 4, 7, 20, 24]. Data-parallel CEP is mainly divided in two categories, namely window-based parallelization [1, 4] and key-based parallelization [1, 7, 20, 24]. The key-based parallelization is limited only to the applications where events could have a key.

Various approximation techniques are frequently used to avoid resource constraints in various domains such as in-network processing [5, 28], distributed graph processing [25], stream processing [21, 23, 29], etc. In fact, load shedding has been proposed by several research groups [3, 13, 14, 16, 21, 23, 29, 30] in the stream processing domain. The idea is to drop events in a way that reduces the system load but still provides the maximum possible quality of result. Hence, the crucial question here is which events (tuples) to drop so the quality of result is not impacted drastically. In [21, 29], the authors assumed that the tuples have different utility values–which reflects their importance and impact on the quality of result. In case of overload, the tuples with low utility values are dropped. The work in [23] proposes to drop tuples that incur higher processing latency where the authors claim that tuples have the same utility value but may have different processing latency. In [22] the authors assume that all tuples have the same utilities and processing latency. They fairly select tuples for drop from different input streams by combining two techniques, stratified sampling and reservoir sampling. In contrast to all these works, in CEP, the utility of events is influenced by other events in the same pattern since CEP systems perform pattern correlation. Hence, we cannot consider only the utility of each event individually but we have to also take into the consideration other events in the pattern. Moreover, the order of events in patterns and in input event streams is important in CEP (e.g., the sequence operator) which is not considered in the above works.

The authors in [12] have formulated the load shedding problem in CEP as a set of different optimization problems. Two types of load shedding are considered by the authors: integral load shedding in which specific event types or patterns are dropped and fractional load shedding where a uniform sampling is used to keep a portion of event types or pattern matches. However, they do not consider the order of events in patterns and in input event streams which is important in CEP. In [27], the authors proposed a load shedding approach, called pSPICE, to drop partial matches instead of events from a CEP operator's internal state where partial matches that have low probabilities to complete and become complex events are dropped. However, as the authors show, pSPICE is outperformed (w.r.t. quality of results) by event dropping strategies if the partial match completion probability is relatively high.

6 Conclusion

In this paper, we proposed a lightweight load shedding framework, called eSPICE, for window-based CEP systems which maintains a given latency bound by dropping events while reducing its adverse impact on the quality of results. eSPICE uses the type and relative position within windows of primitive events to predict their utility values and efficiently drops events from incoming windows. Through extensive evaluations on two real world datasets and a range of popular CEP operators, we show that eSPICE outperforms state-of-the-art load shedders for CEP/stream processing systems. eSPICE successfully maintains the given latency bound while keeping the degradation in quality of results very low at minimum overhead.

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).

References

[1] [n. d.]. Apache Storm. http://storm.apache.org. ([n. d.]). 05.05.2019.

[2] [n. d.]. Google Finance. https://www.google.com/finance. ([n. d.]). 05.05.2019.

[3] B. Babcock, M. Datar, and R. Motwani. 2004. Load shedding for aggregation queries over data streams. In *Proceedings. 20th International Conference on Data Engineering*. 350–361.

[4] Cagri Balkesen, Nihal Dindar, Matthias Wetter, and Nesime Tatbul. 2013. RIP: Run-based Intra-query Parallelism for Scalable Complex Event Processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems (DEBS ’13)*. ACM, New York, NY, USA, 3–14.

[5] Sukanya Bhowmik, Muhammad Adnan Tariq, Alexander Balogh, and Kurt Rothermel. 2017. 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)*.

[6] S. Bhowmik, M. A. Tariq, J. Grunert, D. Srinivasan, and K. Rothermel. 2018. Expressive Content-Based Routing in Software-Defined Networks. *IEEE Transactions on Parallel and Distributed Systems* 29, 11 (Nov 2018), 2460–2477.

[7] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. 2013. Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD ’13)*. ACM, New York, NY, USA, 725–736.

[8] S. Chakravarthy and D. Mishra. 1994. Snoop: An Expressive Event Specification Language for Active Databases. *Data Knowl. Eng.* 14, 1 (Nov. 1994), 1–26.

[9] Gianpaolo Cugola and Alessandro Margara. 2010. TESLA: A Formally Defined Event Specification Language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems (DEBS ’10)*. ACM, New York, NY, USA, 50–61.

[10] Gianpaolo Cugola and Alessandro Margara. 2012. Processing Flows of Information: From Data Stream to Complex Event Processing. *ACM Comput. Surv.* 44, 3, Article 15 (June 2012), 62 pages.

[11] Miyuru Dayarathna and Srinath Perera. 2018. Recent Advancements in Event Processing. *ACM Comput. Surv.* 51, 2, Article 33 (Feb. 2018), 36 pages.

[12] Yeye He, Siddharth Barman, and Jeffrey F. Naughton. 2014. On Load Shedding in Complex Event Processing. In *Proceedings of the 17th International Conference on Database Theory (ICDT ’17)*. Athens, Greece.

[13] Evangelia Kalyvianaki, Marco Fiscato, Theodoros Salonidis, and Peter Pietzuch. 2016. THEMIS: Fairness in Federated Stream Processing Under Overload. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD ’16)*. ACM, New York, NY, USA, 541–553.

[14] N. R. Katsipoulakis, A. Labrinidis, and P. K. Chrysanthis. 2018. Concept-Driven Load Shedding: Reducing Size and Error of Voluminous and Variable Data Streams. In *2018 IEEE International Conference on Big Data (Big Data)*. 418–427.

[15] G. F. Lima, A. Slo, S. Bhowmik, M. Endler, and K. Rothermel. 2018. 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)*. 31–40.

[16] Gurmeet Singh Manku and Rajeev Motwani. 2002. Approximate Frequency Counts over Data Streams. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB ’02)*. VLDB Endowment, 346–357.

[17] Ruben Mayer, Ahmad Slo, Muhammad Adnan Tariq, Kurt Rothermel, Manuel Gräber, and Umakishore Ramachandran. 2017. SPECTRE: Supporting Consumption Policies in Window-based Parallel Complex Event Processing. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware ’17)*. ACM, New York, NY, USA, 161–173.

[18] Ruben Mayer, Muhammad Adnan Tariq, and Kurt Rothermel. 2017. Minimizing Communication Overhead in Window-Based Parallel Complex Event Processing. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems (DEBS ’17)*. ACM, New York, NY, USA, 54–65.

[19] Christopher Mutschler, Holger Ziekow, and Zbigniew Jerzak. 2013. The DEBS 2013 Grand Challenge. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems (DEBS ’13)*. ACM, New York, NY, USA, 289–294.

[20] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. 2010. S4: Distributed Stream Computing Platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*. 170–177.

[21] Chris Olston, Jing Jiang, and Jennifer Widom. 2003. Adaptive Filters for Continuous Queries over Distributed Data Streams. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD ’03)*. ACM, New York, NY, USA, 563–574.

[22] Do Le Quoc, Ruichuan Chen, Pramod Bhatotia, Christof Fetzer, Volker Hilt, and Thorsten Strufe. 2017. StreamApprox: Approximate Computing for Stream Analytics. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware ’17)*. ACM, New York, NY, USA, 185–197.

[23] Nicoló Rivetti, Yann Busnel, and Leonardo Querzoni. 2016. Load-aware Shedding in Stream Processing Systems. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems (DEBS ’16)*. ACM, New York, NY, USA, 61–68.

[24] M.A. Shah, J.M. Hellerstein, S. Chandrasekaran, and M.J. Franklin. 2003. Flux: an adaptive partitioning operator for continuous query systems. In *Data Engineering, 2003. Proceedings. 19th International Conference on*. 25–36.

[25] Zechao Shang and Jeffrey Xu Yu. 2014. Auto-approximation of graph computing. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1833–1844.

[26] Steven S. Skiena. 2008. *Sorting and Searching*. Springer London, London, 103–144.

[27] Ahmad Slo, Sukanya Bhowmik, Albert Flaig, and Kurt Rothermel. 2019. pSPICE: Partial Match Shedding for Complex Event Processing. In *2019 IEEE International Conference on Big Data*.

[28] Muhammad Adnan Tariq, Boris Koldehofe, Sukanya Bhowmik, and Kurt Rothermel. 2014. PLEROMA: A SDN-based High Performance Publish/Subscribe Middleware. In *Proceedings of 15th International Middleware Conference*.

[29] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. 2003. Load Shedding in a Data Stream Manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29 (VLDB ’03)*. VLDB Endowment, 309–320.

[30] Nesime Tatbul and Stan Zdonik. 2006. Window-aware Load Shedding for Aggregation Queries over Data Streams. In *Proceedings of the 32Nd International Conference on Very Large Data Bases (VLDB ’06)*. VLDB Endowment, 799–810.

[31] Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-performance Complex Event Processing over Streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD ’06)*. ACM, New York, NY, USA, 407–418.

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

[33] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. On Complexity and Optimization of Expensive Queries in Complex Event Processing. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD ’14)*. ACM, New York, NY, USA, 217–228.

[34] D. Zimmer. 1999. On the Semantics of Complex Events in Active Database Management Systems. In *Proceedings of the 15th International Conference on Data Engineering (ICDE ’99)*. IEEE Computer Society, Washington, DC, USA.