Load Shedding in Window-Based Complex Event Processing

Von der Fakultät Informatik, Elektrotechnik und
Informationstechnik der Universität Stuttgart
zur Erlangung der Würde eines Doktors der Naturwissenschaften
(Dr. rer. nat.) genehmigte Abhandlung

vorgelegt von

Ahmad Slo

aus Aleppo/Syrien

Hauptberichter:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Mitberichter:	Prof. Dr. Matthias Weidlich
Tag der mündlichen Prüfung:	*31.01.2022*

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart
2022

Acknowledgments

First and foremost, I would like to praise and thank God (Allah), the almighty, for granting me the opportunity to successfully complete my PhD research, and for the other countless blessings in my life.

I would like to express my deep gratitude to my supervisor, Prof. Dr. Kurt Rothermel, for giving me the opportunity to work in his research group and to write my PhD thesis. His valuable comments and feedback were important factors in improving the quality of my PhD thesis. I would also like to thank my post-doc, Dr. Sukanya Bhowmik, for her support and for giving me valuable feedback, especially on my paper write-ups.

Furthermore, I would like to sincerely thank Prof. Dr. Matthias Weidlich for being a part of my PhD committee and reviewing my thesis.

During my doctoral studies, I have had the chance to work and interact with many wonderful colleagues. Especially, I thank my friend and colleague, Dr. Mohamed Abdelaal, for his fruitful discussions and support. Moreover, I wish to thank all my colleagues, especially, Jonathan Falk, Saravana Murthy, Henriette Röger, Zohaib Riaz, Ben Carabelli, Christoph Dibak, Otto Bibartiu, and Johannes Kässinger. I am also grateful to Eva Strähle and Manfred Rasch for their support with paperwork.

Last but not least, I would like to thank my family and friends for their support during my PhD. Especial thanks go to my friends Dr. Houssem Ben Lahmar and Dr. Mohammad Hamad. I express my sincere gratitude and immense respect for my parent, Hasan Slo (may Allah bless his soul) and Khadija Al-Mousa, and my siblings for their extraordinary role in my life. I greatly thank my wife, Fadwa, for her continuous support and encouragement throughout my doctoral journey. I am grateful to my small lovely daughter, Leen, who was the main source to relieve the stress during home office and Corona times.

Contents

List of Figures

List of Tables

List of Algorithms

Abstract

Complex event processing (CEP) is a powerful paradigm to detect patterns in continuous input event streams. The application area of CEP is very broad, e.g., transportation, stock market, network monitoring, game analytics, retail management, etc. A CEP operator performs pattern matching by correlating input events to detect important situations (called complex events). The criticality of detected 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, the detected complex events might be less critical. As a result, these applications might tolerate imprecise detection or loss of some complex events.

In many applications, the rate of input events is high and exceeds the processing capacity of CEP operators. Moreover, for many applications, it is important to detect complex events within a certain latency bound, where the late detected complex events might become useless. 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. Load shedding reduces the overload on a CEP operator by either dropping events from the operator's input event stream or dropping partial matches (short PM) from the operator's internal state. That results in decreasing the number of queued events and in increasing the operator processing rate, hence enabling the operator to maintain the given latency bound. Of course, dropping might adversely impact the quality of results (QoR). Therefore, it is crucial to shed load in a way that has a low impact on QoR.

There exists only limited work on load shedding in the CEP domain. Therefore, in this thesis, we aim to realize a load shedding library that contains several load shedding approaches for CEP systems. Our shedding approaches drop events and PMs, shed events on different granularity levels, and use several features to predict the importance/utility of events and PMs. More specifically, our contributions are as follows.

At first, we precisely define the quality of results (QoR) using real-world examples

and different pattern matching semantics defined in the CEP domain. Secondly, we propose a load shedding approach (called pSPICE) that drops PMs to maintain a given latency bound. pSPICE uses the Markov chain and Markov reward process to predict the utility of PMs. Moreover, pSPICE adaptively calculates the number of PMs that must be dropped to maintain the given latency bound.

In our third and fourth contributions, we develop two load shedding approaches that are called eSPICE and hSPICE. eSPICE drops events from windows to maintain the given latency bound. While hSPICE drops events from windows and PMs to maintain the given latency bound. Both approaches use a probabilistic model to predict the event utilities. Moreover, in both approaches, we provide algorithms that predict utility thresholds to drop the needed number of events. Additionally, in eSPICE, we develop an algorithm that adaptively calculates the number of events that must be dropped to maintain the given latency bound.

Finally, we propose a load shedding approach (called gSPICE) that drops events from the input event stream and from windows to maintain the given latency bound. gSPICE also predicts the event utilities using a probabilistic model. Moreover, to efficiently store the event utilities, we develop a data structure that depends on the Zobrist hashing. Furthermore, gSPICE uses well-known machine learning approaches, e.g., decision trees or random forests, to estimate event utilities.

We extensively evaluate our proposed load shedding approaches on several real-world and synthetic datasets using a wide range of CEP queries.

Deutsche Zusammenfassung

Complex Event Processing (CEP) oder komplexe Ereignisverarbeitung ist ein leistungsfähiges Paradigma zur Erkennung von Ereignismustern in kontinuierlichen Eingangsereignisströmen. CEP ist breit anwendbar, z.B. in den Bereichen Verkehr, Börsenhandel, Netzwerküberwachung, Spielanalyse, Einzelhandelsmanagement, usw. Ein CEP-Operator führt einen Mustervergleich durch, indem er Eingangsereignisse korreliert, um wichtige Situationen (komplexe Ereignisse genannt, engl. complex events) zu erkennen. Die Kritikalität der erkannten komplexen Ereignisse hängt von der Anwendung ab. In Betrugserkennungssystemen in Banken können die erkannten komplexen Ereignisse darauf hinweisen, dass ein Betrüger versucht, Geld vom Konto des Opfers abzuheben. Natürlich sind die komplexen Ereignisse in dieser Anwendung kritisch. Auf der anderen Seite sind in Anwendungen wie Netzwerküberwachung, Fußballanalyse und Verkehr die erkannten komplexen Ereignisse weniger kritisch. Folglich können diese Anwendungen ungenaue Erkennung oder den Verlust einiger komplexer Ereignisse tolerieren.

In vielen Anwendungen ist die Rate der Eingangsereignisse hoch und übersteigt die Verarbeitungskapazität der CEP-Operatoren. Außerdem ist es für viele Anwendungen wichtig, komplexe Ereignisse innerhalb einer bestimmter Latenzschranke zu erkennen, wobei verspätet erkannten komplexen Ereignisse unbrauchbar werden könnten. Für CEP-Anwendungen, die eine ungenaue Erkennung von komplexen Ereignissen tolerieren und über begrenzte Verarbeitungsressourcen verfügen, bietet die Verwendung von Lastabwurf (engl. load shedding) eine Möglichkeit zum Einhalten der vorgegebenen Latenzschranke. Lastabwurf reduziert die Überlastung eines CEP-Operators indem entweder Ereignisse aus dem Eingangsereignisstrom des Operators verworfen werden oder partielle Übereinstimmungen (engl. partial matches, Abk. PM) aus dem internen Zustand des Operators entfernt werden. Das führt zu einer Verringerung der Anzahl von Ereignissen in der Warteschlange und erhöht die Verarbeitungsrate des Operators, wodurch der Operator in die Lage versetzt wird, die vorgegebene Latenzschranke einzuhalten. Natürlich kann das Verwerfen von Ereignissen die Qualität der Ergebnisse (engl. quality of results, Abk. QoR) negativ beeinträchtigen. Daher ist es entscheidend, die Last so abzuwerfen, dass sich eine geringe Auswirkung auf die QoR ergibt

Es gibt nur wenige Arbeiten zum Lastabwurf im CEP-Bereich. Daher wird in dieser Arbeit eine Lastabwurf-Bibliothek realisiert, die verschiedene Lastabwurf-Ansätze für CEP-Systeme enthält. Unsere Ansätze können Ereignisse und PMs verwerfen, wobei wir verschiedene Merkmale zur Vorhersage der Wichtigkeit/Nützlichkeit von Ereignissen und PMs verwenden. Außerdem, werden Ereignisse auf verschiedenen Ebenen verworfen. Im Einzelnen sind unsere Beiträge wie folgt.

Erstens definieren wir die Ergebnisqualität (QoR) anhand von Beispielen aus der realen Welt und verschiedenen Ereignismustersemantiken, die in der CEP-Domäne definiert sind. Zweitens schlagen wir einen Lastabwurf-Ansatz (genannt pSPICE) vor, der PMs auslässt, um eine gegebene Latenzschranke einzuhalten. pSPICE verwendet die Markov-Kette und den Markov-Reward-Prozess, um den Nutzen von PMs vorherzusagen. Außerdem berechnet pSPICE adaptiv die Anzahl der PMs, die verworfen werden müssen, um die vorgegebene Latenzschranke einzuhalten.

In unserem dritten und vierten Beitrag entwickeln wir zwei Ansätze zum Lastabwurf, die eSPICE und hSPICE genannt werden. eSPICE verwirft Ereignisse aus Fenstern, um die gegebene Latenzschranke einzuhalten, während hSPICE Ereignisse von Fenstern und PMs auslässt, um die vorgegebene Latenzschranke einzuhalten. Beide Ansätze verwenden ein probabilistisches Modell zur Vorhersage der Nutzung der Ereignisse. Darüber hinaus stellen wir in beiden Ansätzen Algorithmen zur Verfügung, die einen Nutzschwellwert vorhersagen, anhand dessen die erforderliche Ereignisanzahl verworfen wird. Zusätzlich entwickeln wir in eSPICE einen Algorithmus, der adaptiv die Anzahl der Ereignisse berechnet, die verworfen werden müssen, um die die gegebene Latenzschranke einzuhalten.

Schließlich schlagen wir einen Lastabwurf-Ansatz (genannt gSPICE) vor, der Ereignisse aus dem Eingangsereignisstrom und aus Fenstern entfernt, um die vorgegebene Latenzschranke einzuhalten. gSPICE prognostiziert auch die Nutzung der Ereignisse unter Verwendung eines probabilistischen Modells. Zur effizienten Speicherung des Nutzens der Ereignisse entwickeln wir eine Datenstruktur, die auf dem Zobrist-Hashing basiert. Darüber hinaus verwendet gSPICE bekannte Ansätze des maschinellen Lernens, z.B. Decision Trees oder Random Forests, um Ereignisnutzwerte zu schätzen.

Wir evaluieren unsere vorgeschlagenen Lastabwurf-Ansätze ausgiebig auf mehreren realen und synthetischen Datensätzen mit einer breiten Palette von CEP-Abfragen.

Chapter 1

Introduction

Generally, most applications rely on high-level data streams, referred to as events. An event is the basic data element of many contemporary important applications, e.g., finance, transportation, fraud detection, stock analysis, smart home, smarts cities, heathcare, environmental monitoring, and network monitoring applications. For example, an event might represent the change in the temperature of equipment or the stock quote of a company. Events emanate from sensors, social media, and various other sources, forming an event stream. For an application, the occurrence of events might indicate the occurrence of application-interesting situations, where this occurrence requires the application to take suitable actions.

In this context, complex event processing (CEP) is an established paradigm used to detect the occurrence of important situations by processing the input event streams [Luc01; DGP07; WDR06; GJS92; CM94]. CEP systems extract high-level information from the low level data elements (i.e., events), where a CEP system correlates events in the input event streams to detect important situations (a.k.a. complex events). CEP acquires a considerable market share, where it is expected to reach USD 10.79 billion by 2023 [Upd18]. The powerful features of CEP open the door for several applications in different domains, e.g., transportation, stock market, network monitoring, game analytics, retail management, and fraud detection [Zac+15; May+17; Bal+13; OJW03; MZJ13; WDR06; Art+17]. The criticality of detected complex events mainly 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 [Art+17]. On the other hand, in applications like network monitoring, soccer analysis, and transportation [OJW03; SBR19; Slo+19], the detected complex events are less critical. As a result, these applications might tolerate imprecise detection or loss of some complex events. Moreover, in many applications, complex events must be detected within a given latency bound to enable the application to take suitable actions at the right time. However, if the rate of input events exceeds the processing capacity of the CEP system, the input

events queue up, and the detection latency of complex events increases, possibly resulting in violation of the given latency bound.

The main objective of this thesis is to empower CEP systems to prevent the violation of the given latency bound during overload situations. To this end, this thesis introduces several load shedding strategies for CEP systems. In the rest of this chapter, we first present an overview of CEP systems. Then, we motivate our work by explaining the overload and latency challenges. Afterward, we define the research gaps and highlight our main contributions in this thesis. Finally, we conclude this chapter with a brief outline of the thesis.

1.1 Complex Event Processing

In general, a CEP system consists of a set of operators that are interconnected in the form of a directed acyclic graph (DAG), constituting the CEP operator graph [Che+03; McC+13; Neu+10; Kol+12; SMMP09]. Each CEP operator correlates events in the input event streams to detect complex events. The event correlation (a.k.a. pattern matching) is typically performed in accordance with predefined CEP patterns. There exist several event query languages, e.g., Snoop [CM94], SASE [WDR06], and TESLA [CM10], that are used to define queries in CEP, where a *query* corresponds to a *pattern*. The event query languages contain several predefined event operators (e.g., sequence, negation, any operators) that help in defining CEP patterns. In CEP applications, an event in the input event stream represents the basic data element. For example, an event might contain information on the stock quote of a company in a stock analysis application or the position of a bus in a transportation application. While complex events, for example, might provide high-level information on stock companies that influence each other or the occurrence of abnormal traffic in the above two applications, respectively.

In CEP systems, the input event stream is continuous and infinite. Therefore, it is common in CEP to correlate together only events that occur within a certain interval (we refer to this interval as a window). The window represents a temporal constraint in CEP systems. This CEP model is known as *window-based* CEP, where the input event stream is partitioned into windows of events [AC04; May18; PS06; Bal+13; Lim+18]. The events within a certain window are matched by a CEP operator to detect complex events, where windows might overlap. Therefore, an event in the input event stream might belong to multiple windows at the same time. While correlating events within a window, a part of a pattern might be matched. This matched part of the pattern is called a partial match (short PM). Within a window, at a certain point in time, there might exist many open PMs, where every incoming event in the window is matched with these open PMs. In CEP systems, PMs represent an important part of the internal state of a CEP operator. A partial match may complete and become a complex event if the full pattern is matched. As a result, in the window-based CEP, the matching of events

is performed on three granularity levels: stream, window, and PM granularities. The stream represents the coarsest granularity, while the PM represents the finest granularity.

The following example shows how a pattern is defined and detected in CEP. In a traffic monitoring system [Zac+15], if more than one bus gets delayed at the same bus stop, it might indicate an abnormal traffic situation, e.g., an accident. To detect the abnormal behavior, a traffic analyst formulates the following query q using SASE-like event query language [WDR06]:

pattern **seq** $(A; B)$
  where $A.delay > x$ minutes **and** $B.delay > x$ minutes
    and $A.stop = B.stop$
  within 5 minutes

The query q detects abnormal traffic, i.e., a complex event, if a bus A gets delayed on a specific bus stop and the following bus B within 5 minutes (window length/size) from bus A also gets delayed at the same bus stop.

Assume that a window w contains the following three events: A_1, A_2, and B_3, where X_i represents the event of bus X at position i in window w. We refer to A and B as event types. If the bus event A_1 indicates that bus A gets delayed, a new PM γ_1 is opened with event A_1. We refer to event A_1 as the event that contributes to PM γ_1. Similarly, in window w, a new PM γ_2 is opened with event A_2 if the bus event A_2 also indicates that the bus A gets delayed. If the bus event B_3 indicates that the subsequent bus B is also delayed at the same stop as A_1 and A_2, this might result in detecting two complex events. The first complex event $cplx_1 = (A_1, B_3)$ is detected as a result of matching event B_3 with event A_1 (i.e., with PM γ_1), and the second complex event $cplx_2 = (A_2, B_3)$ is detected as a result of matching event B_3 with event A_2 (i.e., with PM γ_2). We refer to events A_1, A_2, and B_3 in complex events $cplx_1$ and $cplx_2$ as the events that contribute to the complex events $cplx_1$ and $cplx_2$.

However, in this example, it is unclear whether event B_3 should match with only event A_1, only event A_2, or with both events A_1 and A_2. Moreover, in the above example, event B_3 contributes to two complex events, i.e., $cplx_1$ and $cplx_2$. Hence, it is unclear whether an event is allowed to be used in detecting multiple complex events, i.e., to contribute to multiple complex events. Therefore, CEP researchers have introduced *selection policies* that define which events must be matched together and *consumption policies* that define whether an event may contribute to more than one complex event. In CEP systems, there exist mainly four selection policies (*first*, *last*, *each*, and *cumulative*) and two consumption policies (*zero* and *consumed*) [CM94; ZU99; CM10]. In Chapter 2, we discuss in more detail windows, PMs, how pattern matching is performed in CEP systems, and the selection and consumption policies.

1.2 Motivation and Research Gaps

As mentioned above, the focus of this thesis is on handling overloaded CEP systems to maintain a given latency bound. Latency is a crucial factor in plenty of paramount CEP applications. A high complex event detection latency might result in losing lives or money. For example, in the case of a vehicle accident, the percentage of lives saved increases by 6% if the accident detection latency decreases by one minute [Eva96]. Moreover, in [ABO20], the authors show that exceeding latency bound in trading on global stock markets costs the investors approximately USD 5 billion per year.

In many CEP applications, e.g., network monitoring, traffic monitoring, stock market [OJW03; Zac+15; Bal+13], the volume of the input event stream is typically high where it is not feasible to process the incoming events on a single machine. Moreover, the detection latency of complex events is significantly important, where the detected complex events might become useless if they are not detected within a certain latency bound [Quo+17; CF+13; RBR19]. To process such huge input event streams and maintain a given latency bound, a well-known solution in the CEP domain is to use distribution and parallelization, where CEP operators are distributed on multiple compute nodes. Moreover, each CEP operator runs on one or more compute nodes [CF+13; Neu+10; May+17; Bal+13; ZR10; Zac+15; MKR15; ZR10]. However, in many applications, the volume of the input event stream is not stable and fluctuates over time [KLC18; RBQ16]. Therefore, it is not trivial to know the number of necessary compute nodes in advance. Hence, either the number of compute nodes should be over-provisioned, which introduces additional cost, or the number of compute nodes can be adapted elastically as proposed by many researchers [CF+13; Neu+10; Zac+15; MKR15; ZR10]. However, adapting the parallelization degree in the case of short input spikes introduces a high performance overhead [KLC18]. Moreover, resources might be limited for several reasons including: 1) limited monetary budget, and 2) limited compute resources if operators run in private clouds due to security or response time reasons.

Load Shedding in CEP. Another solution to process large input event streams while maintaining a given latency bound is to reduce the processing latency of events in CEP operators, which in turn increases the operator processing rate, hence enabling to maintain the given latency bound. In a CEP operator, the processing latency of an event e represents the time an operator needs to process (i.e., match) the event e with open PMs within all windows. Hence, the event processing latency mainly depends on the number of open PMs within the operator [RLR16; Bal+13]. Therefore, to reduce the processing latency of an event e, we need to reduce the number of PMs with which the event e is matched within an operator. This can be done either by dropping events or PMs (a.k.a. load shedding). As expected, load shedding may negatively impact the quality of results (short 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. As a result, for CEP applications that tolerate imprecise detection of complex events and have limited processing resources, one effective way to keep the given latency bound is through intelligently shedding the excessive load.

In CEP systems, to maintain a given latency bound and minimize the adverse impact of shedding on QoR, a load shedding approach should perform the following three main tasks [Slo+19; SBR19; Tat+03]: 1) Deciding on when to drop events or PMs, 2) Calculating the number of events or PMs that are needed to be dropped, and 3) Deciding on which events or PMs to drop. The third task (i.e., deciding on which events or PMs to drop) is the most crucial task since it directly influences QoR. One way to tackle this task is by assigning utilities to events/PMs where the utilities reflect the importance of events/PMs, w.r.t. QoR. The higher is the importance of an event/PM for QoR, the higher is its utility. As a result, if there is a need to shed load, events/PMs with the lowest utilities are dropped, hence reducing the negative impact of dropping on QoR. The utility of an event depends mainly on the detected complex events to which the event contributes. Similarly, the utility of a PM depends on the complex events that are detected when the PM completes. Hence, the detected complex events might be used as a feature to assign utilities to events/PMs. The higher is the number of detected complex events to which an event contributes or a PM becomes, the higher is the utility of the event/PM.

In CEP systems, complex events can be only identified after processing (i.e., matching) events in the input event stream with PMs within CEP operators. However, the CEP systems need to assign utilities to events and PMs prior to processing events with PMs, hence being able to drop events/PMs with the lowest utilities in overload cases, maintaining the given latency bound, and reducing the drop impact on QoR. That means that to assign utilities to events/PMs, we must depend on features other than the detected complex events. An efficient load shedding approach must use appropriate features that help in precisely predicting the event/PM utilities. In CEP systems, there exist many features that may be used to predict the events/PM utilities. These features may originate from events/PMs themselves or from the context in which the events/PMs exist. The event type and the PM state (i.e., the progress of the PM) are examples of features that originate from events and PMs themselves, respectively. While the event positions within a window and the recently occurred events in a window represent examples of features that originate from the context.

Besides precisely predicting event/PM utilities, another crucial factor influencing 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 is used to make the shedding decisions. Such overhead results in reducing the available processing power to perform pattern matching, thus adversely impacting QoR. There exists a trade-off between precisely predicting the event/PM utilities and the load

shedding overhead. As we showed above, an event in the input event stream is processed (i.e., matched) on three different granularity levels: the stream, the window, and the PM levels. Accordingly, event dropping might also be performed on these three different granularity levels. 1) Dropping an event from the event stream is equivalent to drop the event from all windows and PMs within an operator. 2) Dropping an event from a window is equivalent to dropping the event from all PMs within the window, which in turn represents a subset of all PMs within an operator. 3) Dropping an event from a PM means that the event is dropped from an individual PM within an operator.

Dropping events on the stream level (i.e., the coarsest granularity) requires taking the shedding decision once for each event in the event stream. While dropping events from windows and PMs might demand to take the shedding decision multiple times for each event in the input event stream. Hence, dropping events on the stream level might impose lower load shedding overhead compared to dropping events on window and PM granularities. However, event utilities on the stream level might not be precisely predicted compared to predicting the event utilities on window and PM granularities. Dropping a PM means that the PM is removed from the operator's internal state, hence no events in the window will be matched with this dropped PM. Therefore, dropping PMs also represents dropping on a coarser granularity compared to dropping events from PMs. Besides the drop granularity, the chosen features to predict the utility of events/PMs might also influence the overhead of a load shedding approach. Therefore, we must select those features that can predict the utility of events/PMs with considerable accuracy and a tolerable overhead.

As a result, we classify the load shedding approaches according to the following three characteristics: 1) Depending on whether the internal state of a CEP operator is revealed (i.e., a white-box or black-box CEP operator). 2) Depending on whether events or PMs are dropped, i.e., shedding is performed on events or PM granularities. 3) Depending on whether events are dropped from the stream, window, or PMs (in the case of dropping events). Figure 1.1 depicts a classification of load shedding approaches in CEP. In the figure, a load shedding approach might be either a black-box or a white-box approach. Moreover, the shedding may be performed on two granularities: event and PM granularities. In a black-box approach, only events might be dropped since PMs are not revealed by CEP operators, i.e., PMs are not accessible. Moreover, dropping events in a black-box approach might be performed only on stream and window granularities. As PMs are not revealed, dropping events on PM granularity is not possible. In a white box load shedding approach, we may shed PMs since the operator's internal state is exposed. Additionally, we may drop events on all granularity levels, i.e., stream, window, and PM levels.

State-of-the-art. Load shedding has been extensively studied in the stream processing domain [Tat+03; TZ06; RBQ16; OJW03; TBL08; SW04; JMR05]. The queries in this domain are mostly aggregations, min, max, and simple binary equi-joins. There-

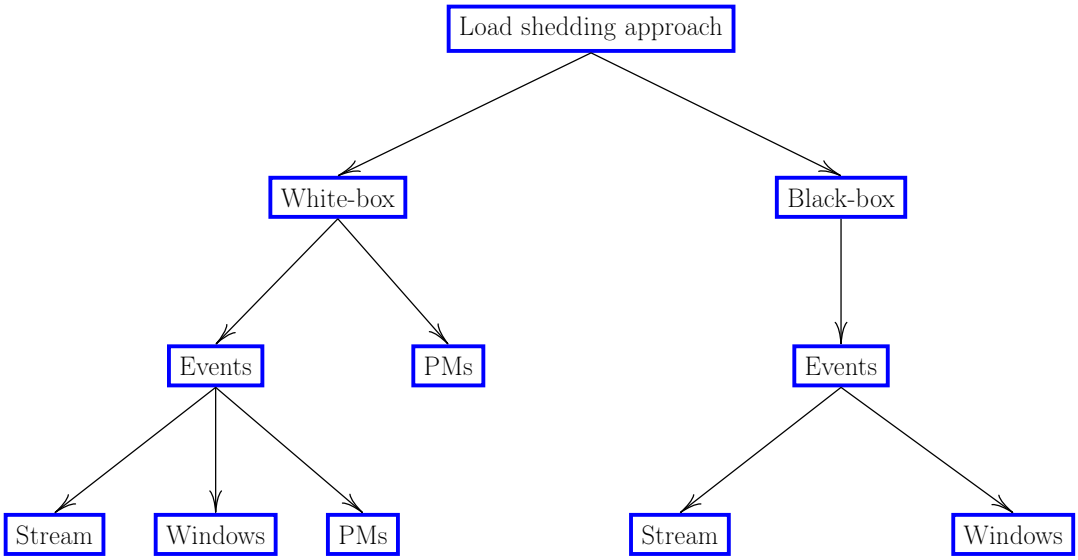


Figure 1.1: Classification of load shedding approaches in CEP.

fore, researchers in the stream processing domain propose load shedding approaches that mainly assign utilities to tuples individually without taking into consideration the dependency between tuples. However, patterns in the CEP domain are different and more complex than the used queries in the stream processing domain. In CEP systems, a pattern can be viewed as multi-relational non-equi-joins with temporal constraints [HBN14]. Moreover, there exists a dependency between events in a pattern. Additionally, CEP has many selection and consumption policies (i.e., match semantics) [CM94; ZU99; CM10]. Using different selection and consumption policies in a pattern might result in detecting a different set of complex events with the same input event stream. These settings considerably increase the complexity of CEP patterns and complicate assigning utilities to events and PMs. Therefore, the load shedding approaches proposed in the stream processing domain are not suitable for the CEP domain.

So far, there exists only a little work on load shedding in CEP [HBN14; ZVHW20]. In [HBN14], the authors propose a black-box load shedding approach for CEP systems where their approach drops events from the input event stream of a CEP operator. The approach assigns utilities to events depending on the dependency between events in patterns and the distribution of events in the input event stream and accordingly shed events. However, they do not consider the order of events in patterns, which is important in CEP as in sequence and negation operators [Liu+09; CGB11; AC06]. In [ZVHW20], the authors propose a white-box load shedding approach that drops both events and PMs where events and PMs with the lowest utilities are dropped. The events are dropped on the stream granularity. A PM is assigned a utility depending on its progress and the number of remaining events in the window. The utility of events in [ZVHW20] depends on the PMs which these events contribute to. Events that contribute to PMs with low utilities are considered to have also 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 [Agr+08]– which is equivalent to *each* selection policy and *zero* consumption policy [CM94; WDR06]– and it does not support the negation event operator.

As a result, the available works on load shedding in CEP are very narrow and have many limitations. Therefore, there is a need to develop new shedding approaches for CEP systems covering large classes of load shedding approaches (cf. Figure 1.1). The proposed approaches should perform load shedding on different granularity levels and work with black-box or white-box CEP operators. Moreover, they should be generic, w.r.t. supporting CEP operators and pattern matching semantics (i.e., selection and consumption policies). Furthermore, since the load shedding overhead has a considerable impact on the effectiveness of load shedding approaches, the developed load shedding approaches are envisioned to have low overhead.

1.3 Contributions

The goal of this thesis is to develop concepts and algorithms that enable load shedding in CEP systems and empower a CEP operator, in overload cases, to maintain a given latency bound when resources are limited. To this end, we developed a load shedding library for CEP systems. The library contains four main load shedding approaches that cover all load shedding classes shown in Figure 1.1, except shedding events on the stream level while using a white-box operator. Our proposed load shedding approaches enable CEP operators to maintain the given latency bound while minimizing the drop impact on QoR. Our shedding approaches implicitly consider the dependency between the events in patterns and in the input event stream. Moreover, they are not restricted to specific CEP event operators or selection and consumption policies, where they support all common CEP event operators and the selection and consumption policies.

In the following, we list the contributions of this PhD thesis in more detail. These contributions are based mainly on work performed and published as part of the PhD thesis [Slo+19; SBR19; SBR20a; SBR20b], where the scientific work contributed by the author of this thesis was about 70%, 70%, 80%, and 85%, respectively.

1. We provide two ways to define the quality of results (QoR), namely strict and relaxed QoR [SBR20b]. Moreover, we support our definitions by example applications from the real-world.

2. A white-box lightweight load shedding approach (called pSPICE) that drops PMs in overload cases to maintain a given latency bound [Slo+19]. pSPICE uses the Markov chain and Markov reward process to predict the utility of PMs where the utility depends on the PM state (i.e., the PM progress) and the remaining events in the window. Moreover, we develop an algorithm that decides when to drop PMs

and estimates how many PMs to drop from a CEP operator to maintain the given latency bound.

3. eSPICE, a black-box lightweight load shedding approach that, in overload cases, drops events from windows to maintain a given latency bound [SBR19]. eSPICE uses a probabilistic model to predict the event utilities where the utility of an event depends on two features: event type and the relative position of the event in the window. eSPICE is also a lightweight load shedding approach. Moreover, we develop an algorithm to estimate the number of events to drop in order to maintain the given latency bound. Furthermore, we provide an algorithm to predict a utility threshold that enables eSPICE to perform shedding in a lightweight way.

4. An efficient load shedding approach (called hSPICE) that drops events either from PMs or from windows in overload cases to maintain a given latency bound [SBR20a; SBR20b]. hSPICE is a white-box approach that performs event shedding on two granularity levels: window and PM levels. hSPICE uses a probabilistic model to predict the event utilities for PMs within windows where the utility of an event for a PM depends on the following features: the type and position of the event within the window and the state of the PM. Moreover, we provide an algorithm to estimate the number of events to drop to maintain the given latency bound.

5. A black-box shedding approach (called gSPICE) that drops events either from windows or from the input event stream in overload cases to maintain a given latency bound. Hence, gSPICE drops events on two granularity levels: the window and the stream levels. gSPICE uses a probabilistic model to predict the event utilities, where the utility of an event depends on the following features: the type and the relative position of the event within the window, the frequency of event types in the predecessor pane of the event, and the event content (i.e., the event actual data). The predecessor pane of event e represents the sequence of events that occur before event e in the input event stream. gSPICE uses the Zobrist hashing [Zob90] to efficiently store the predicted event utilities. Moreover, gSPICE uses well-known machine learning approaches, e.g., decision trees or random forests, to estimate event utilities. The author of this thesis contributed around 85% of the scientific content of this approach.

We have implemented all of these load shedding approaches by extending a prototype CEP framework that is implemented using Java. Moreover, we have extensively studied their performance with a representative set of CEP queries and several real-world and synthetic datasets.

The above listed contributions and the associated publications are the results of the research work conducted at the University of Stuttgart under the project "Parallel

complex event processing to meet probabilistic latency bounds II (Precept II)". Precept II is funded by the German Research Foundation "Deutsche Forschungsgemeinschaft (DFG)" with grant numbers BH 154/1-2 and RO 1086/19-2.

1.4 Structure

The rest of this thesis is structured as follows. Chapter 2 introduces the basic foundations in this thesis, describing the system model and assumptions. Moreover, it precisely defines the quality of results (QoR) and the problem statement. Chapter 3 presents pSPICE, discussing how the utility of PMs, how the number of required PMs to drop are calculated, and how PM shedding is performed. In Chapter 4, eSPICE is presented, where we explain how eSPICE predicts the event utilities, computes the number of events to drop, and estimates a utility threshold. Chapter 5 details hSPICE and compares its performance with the performance of pSPICE and eSPICE. Chapter 6 presents gSPICE and shows the impact of using several features on the utility prediction and the load shedding overhead. Moreover, it compares the performance of all proposed load shedding approaches. Finally, in Chapter 7, we discuss the related work, and in Chapter 8, we conclude the thesis with a summary of our contributions and a discussion on the possible future work.

Chapter 2

Foundations and Problem Statement

In this chapter, we first present the system model used in this thesis. Then, we provide a detailed definition of the quality of results. Finally, we precisely state the problem that is solved in this work.

2.1 System Model

Operator Graph. A CEP system is modeled as a directed acyclic graph (DAG) where the vertices represent a set of event producers, event consumers, and operators. The edges represent the flow of event streams. Event producers generate primitive events, while event consumers consume the detected complex events. Operators correlate incoming input events using defined patterns to detect complex events. Figure 2.1 depicts an exemplary CEP operator graph that consists of three event producers, three operators, and two event consumers. In the figure, operator op_1 receives input events from event producers pr_1 and pr_2, while operator op_2 receives input events from event producer pr_3. Operators op_1 and op_2 correlate incoming input events and forward the correlation results to operator op_3. Operator op_3, in turn, correlates events in its incoming input event streams and forwards the detected complex events (if any) to event consumers cr_1 and cr_2.

In the following, we explain in detail the basic elements and components of the CEP operator graph.

Primitive Event. A primitive event is the basic data element in CEP systems that represents the occurrence of an application-related situation. A primitive event (or simply, event) is atomic (i.e., happens completely or not at all) and happens at a certain point in time [GJS92; ZU99; CM94]. Examples of primitive events are the following: an RFID tag in retail management applications, the occurrence of an operation (e.g.,

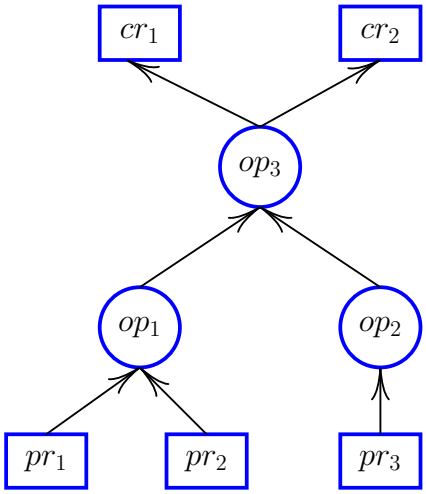


Figure 2.1: An example of a CEP operator graph.

insert, update, delete, etc.) in database applications, information on a player in soccer applications, a change in a bus location in transportation applications, or a change in the stock quote of a company in stock market applications. An event consists of meta-data and a set of attribute-value pairs. The meta-data contains event type and timestamp, while the attribute-value pairs represent the actual event data (i.e., event content). The set of all event types is denoted by $\mathbb{T}$. For example, the type (denoted by $T_e \in \mathbb{T}$) of an 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. Event timestamp represents the point in time when the event occurred where each event e is assigned a timestamp. Moreover, the set of all event attributes of an event e is denoted by $\mathbb{E}_e$. An attribute (denoted by $E_e \in \mathbb{E}$) of an event e might represent a stock quote, a player position, or a bus location in the above applications. An instance of an event represents the occurrence of a specific event type at a certain point in time. For example, in a stock market application, the stock quote of the IBM company might continuously change over time, where an event instance of type IBM is generated for each change of the IBM stock quote.

Complex Event. A complex event is defined similarly to a primitive event where it consists of meta-data and attribute-value pairs. However, a complex event is constituted by correlating two or more events (primitive or complex events) by a CEP operator. Moreover, the attribute-value pairs in a complex event might contain information on the events from which the complex event is detected. For example, in a stock market application, assume that a complex event is detected if the stock quote of company A and the stock quote of company B changes by more than 5%. A detected complex event might contain the amount of change in the stock quote of both companies, or it even might include the stock event of both companies themselves. The attribute-value pairs might also contain any additional information needed by the application. For instance,

in the above example, it might be necessary that the detected complex events must also include the sum of the stock quotes of both companies.

Event Producer. An event producer emits primitive events. Examples of event producers are the following: sensors, RFID readers, social networks, stock exchanges, applications, etc.

Event Consumer. An event consumer receives the detected complex events. Event consumers might be applications, machines, humans, etc.

Event Stream. An event stream is an infinite sequence of events. In the CEP operator graph, the flow of events (primitive or complex) between any two vertices represents an event stream. As an operator might have more than one input event stream, events in the incoming input event streams of an operator merge into a single event stream. Events must be merged in a deterministic order since events order is crucial in CEP, e.g., in the sequence and negation event operators [Liu+09; CGB11; AC06]. Events emitted by an event producer are ordered using event timestamps. We assume that any two events from the same event producer have different timestamps. Moreover, several event streams merge into a single event stream where events in the merged event stream have a total order using the event timestamps and a tie-breaker.

Events in event streams might suffer from different delays due to several types of delay, e.g., transmission and processing delay, which might bring events into a wrong order. This problem is called out-of-order events arrival, a well-known problem in CEP and stream processing domains. In the literature, there exists extensive work on handling the arrival of the out-of-order events, for example, by using heartbeats or slack time [Li+08; MP13; Bri+08; CGM10; Riv+18]. We assume that events in an event stream have a correct order. Additionally, we assume that several input event streams are merged into a single event stream while correctly preserving the total order of events.

Pattern. A CEP pattern defines a set of rules that specify how a certain set of events are correlated. It defines causal dependencies between events, temporal constraints, and conditions on event attributes [Luc01; Liu+09; CM94]. In CEP, events are correlated to check whether they match a defined pattern– we refer to this process as *pattern matching*. As we mentioned in Chapter 1, to define patterns, an event query language might be used, where a *query* corresponds to a *pattern*. Examples of such event languages are Tesla [CM10], Snoop [CM94], and SASE [WDR06]. These languages contain several event operators: sequence, negation, any, Kleene closure, conjunction, disjunction, etc.

Operator. A CEP operator correlates events in its input event streams to detect complex events, following rules defined by patterns, where an operator might match one or more patterns. A CEP operator receives input event streams from its upstream

vertices in the operator graph. As we mentioned above, input event streams of an operator are first merged into a single event stream where the total order is preserved using the event timestamps and a tie-breaker. Then, the operator performs the pattern matching on the merged event streams. Finally, the detected complex events by the operator are forwarded to the downstream vertices in the operator graph that might be other operators or event consumers.

Window. The input event stream of a CEP operator is continuous and infinite. However, in CEP systems, it is common to partition the input event stream into windows of events. This CEP model is also known as *window-based* CEP [AC04; MKR15; PS06; Bal+13; CM10; MM16; Gro+16; AC04]. The window slides over the input event stream, where the new incoming events are continuously added to the window while old events are removed from the window. Windows are opened and closed depending on predicates. The predicates to open and close windows may depend on time (called a time-based predicate), on the number of events (called a count-based predicate), on logical predicates (called a pattern-based predicate), or on a combination of them [MKR15; Gro+16; AC04; Li+05; Bab+02]. The window length may be defined by time (called time-based sliding window), by the number of events (called count-based sliding window), or by logical conditions (called pattern-based sliding window). For example, a window of length 10 seconds or a window of length 1000 events. We refer to the number of events within a window as window size (denoted by ws). Each event within a 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.

We assume that there exists a component called merger-splitter as a predecessor in front of each operator in the operator graph. This component merges the input event streams of an operator into a single event stream while preserving the total event order. Moreover, the merger-splitter partitions the merged input event streams into windows of events using defined predicates. Please note that, for simplicity, this component is not shown in Figure 2.1.

Operator Functionality. To better understand the operator functionally, Figure 2.2 depicts an example of a CEP operator with the merger-splitter component. The figure shows that input event streams (S_1 ... S_m, where $m >= 1$) are first merged into a single input event stream and partitioned into windows of events by the merger-splitter. Windows of events are then pushed to the input queue of the operator. The operator continuously gets windows of events from its input queue and processes them by the *process* function, which performs the actual pattern matching. The output of the pattern matching represents *complex events*. Hence, the operator functionality is represented by the following function: $f : w \rightarrow (c_1, c_2, ...)$, where w represents a window of events and $(c_1, c_2, ...)$ represents an ordered set of complex events. As we mentioned above, an

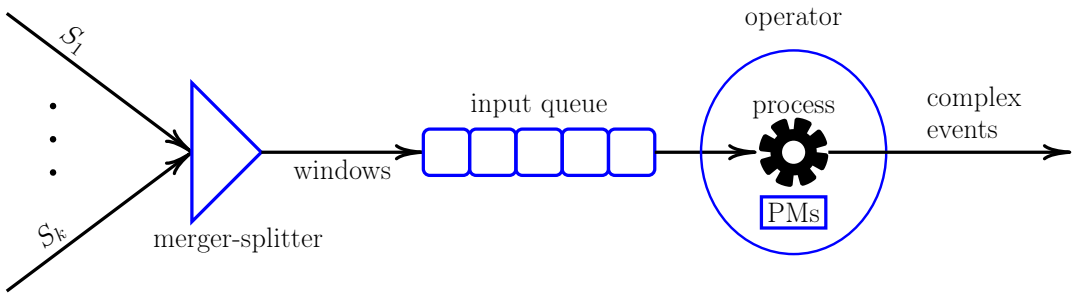


Figure 2.2: An example of a CEP operator with the merger-splitter component.

event might belong to multiple overlapped windows. However, the event is processed *independently* in each window.

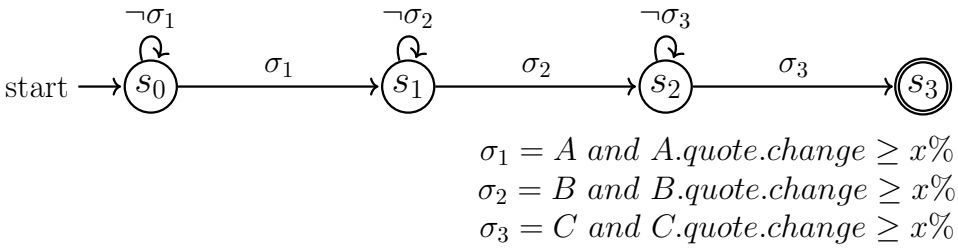
A CEP operator matches one or more patterns (i.e., multi-query). We define the set of patterns that the operator matches as $\mathbb{Q} = \{q_i : 1 \leq i \leq n\}$, where n is the number of patterns. Since patterns might have different importances, each pattern has a weight reflecting its importance. 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.

To clarify the system model, let us introduce the following example.

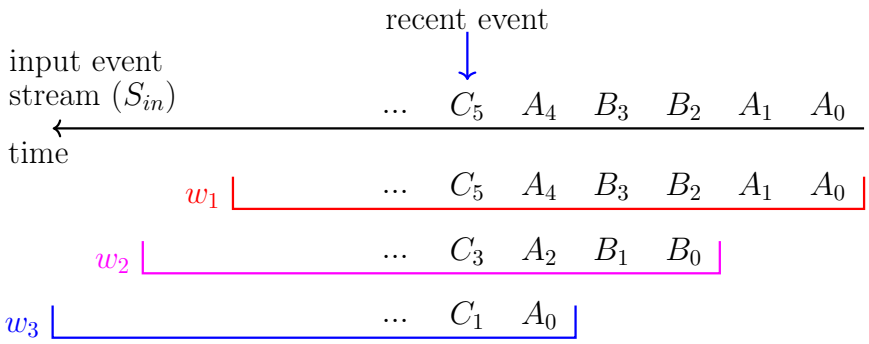
Example 1. In a stock application, an operator matches pattern q, which correlates stock events from three companies. Pattern q is defined as follows: within a window of 8 events (i.e., a count-based sliding window), generate a complex event if a change in the stock quote of company A results in a change in the stock quote of company B, followed by a change in the stock quote of company C– the stock quote of each company should change by at least $x\%$. We may write this pattern as a sequence operator using Snoop event language as follows [CM94]: $q = seq(A; B; C)$. Moreover, we may write the pattern q more precisely in SASE-like event language [WDR06] as follows:

pattern **seq** $(A; B; C)$
where $A.quote, B.quote,$ and $C.quote$ *change by* $\geq$ $x\%$
within 8 events

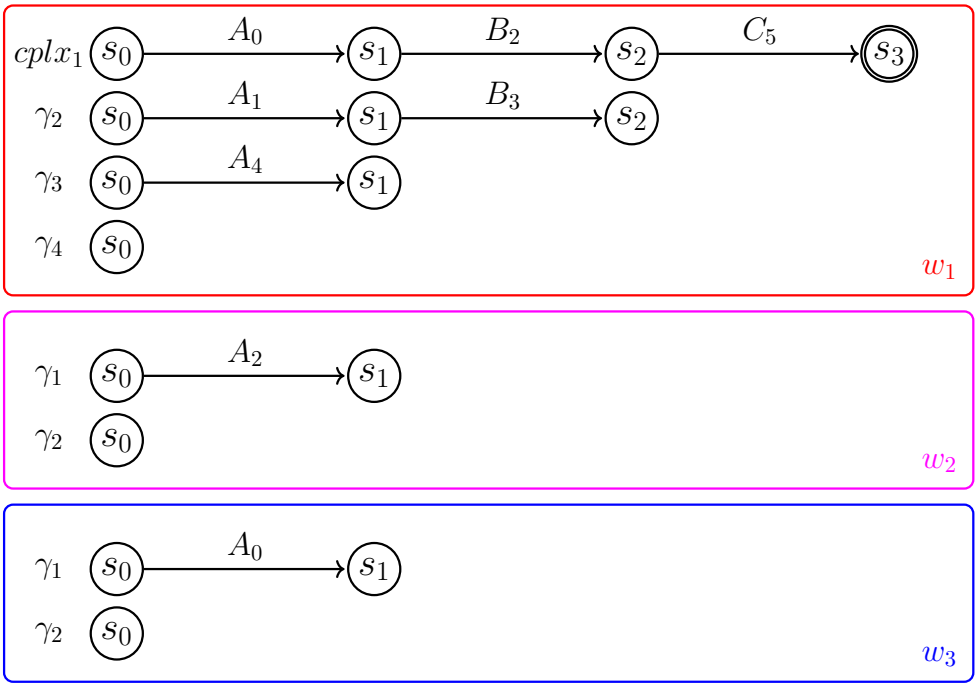
In this example, the operator matches only pattern q. Therefore, the set of patterns that the operator matches is $\mathbb{Q} = \{q\}$. Moreover, in this example, the event type T_e represents the company name, i.e., A, B, or C. Therefore, the set of all event types $\mathbb{T} = \{A, B, C\}$. Furthermore, there exists only one event attribute E_e that is the stock quote. Hence, the set of all event attributes is as follows: $\mathbb{E}_e = \{quote\}$. Assume that a count-based predicate is used to open windows where a window is opened every two events, i.e., the window slide size is two. Figure 2.3 depicts this example. Figure 2.3(a) shows an ordered input event stream (S_{in}) where events in S_{in} are given sequence numbers reflecting their orders. In this example, to simplify the presentation assume



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



(a)



(b)

Figure 2.3: Example 1.

that the change in the stock quote of any event in S_{in} is higher than or equal to $x\%$. Events in S_{in} are instances of the event types A, B, and C. For example, events A_0, B_2, and C_5 in S_{in} are event instances of the event types A, B, and C, respectively. Moreover, the figure shows that the input event stream S_{in} is partitioned into windows where there are three overlapped open windows. As an example to show how the same event may have different positions within different windows, we see that 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. The operator processes an event within all windows to which the event belongs before proceeding to process the next event in the input queue. For example, in Figure 2.3(a), the operator first processes event A_4 of the input event stream (S_{in}) in all open windows, i.e., w_1, w_2, and w_3. Then, it proceeds to process the next event of the input event stream (S_{in}), which is event C_5 in this example.

Finite State Machine (FSM). There exist several methods (a.k.a. computational models) to detect a pattern in CEP, e.g., finite state machine-based methods [May+17; Agr+08; CM10; RLR16; WDR06], tree-based methods [CM94; Cha+94; MM09], string-based methods [Sad+04], and Petri Nets-based methods [GD94]. 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 FSM (cf. Figure 2.3(a)). Please note that our proposed load shedding approaches are agnostic to the used computational model. We will later show how our approaches support other computational models. The set of all possible states $\mathbb{S}_{q_i}$ of pattern $q_i \in \mathbb{Q}$ is defined as follows: $\mathbb{S}_{q_i} = \{s_0, s_1, ..., s_m\}$. In Example 1, pattern q has four states (i.e., $m = 3$) where $\mathbb{S}_q = \{s_0, s_1, s_2, s_3\}$ as shown in Figure 2.3(a). In the figure, s_0 represents the initial state of pattern q and s_3 represents its final state.

Partial Match. Whenever an operator starts to process events within a window, it starts an instance of the FSM of every pattern $q_i \in \mathbb{Q}$ at the initial state. During event processing within a window, an event is matched with the FSM instances of pattern $q_i \in \mathbb{Q}$. The event might cause the FSM 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. Now, let us define it more formally. An instance of the FSM of pattern q_i is called a PM, where the PM completes and becomes a complex event if the FSM 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 $\gamma \subset q_i$ 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 [CM94] (i.e., using the first selection policy and consumed consumption policy for all event types– later in this section, we discuss the selection and consumption policies) and the operator has already processed all available events in all open windows, i.e.,

the operator has processed the last event of type C (C_5 in the input event stream S_{in}) within all windows. Figure 2.3(b) shows the result of pattern matching in all windows. In window w_1, the operator has detected one complex event ($cplx_1 = (A_0, B_2, C_5)$) 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. We refer to events A_0, B_2, C_5 in the complex event $cplx_1$ as events that contribute to complex event $cplx_1$. 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., $B_0 \otimes \gamma_{1s_0}$), 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., $A_2 \in \gamma_{1_{s_0}}$. Please note that in the negation event operator [RLR16; WDR06], 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 these abandoned PMs as completed PMs.

Selection Policy. In window w_1, there exist three instances of the event type A (i.e., A_0, A_1, A_4), two instances of event type B (i.e., B_2, B_3), and one instance of event type C (i.e., C_5). Therefore, it is unclear which instances of the event types A, B, and C should be matched with each other within window w_1. The generated complex events could be any combinations of these event instances. To precisely define which event instances should participate in detecting complex events, in CEP, the *selection policy* has been introduced [CM94; ZU99; CM10]. 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, within window w_1, a detected complex event using the *first* selection policy for all event types might be $cplx = (A_0, B_2, C_5)$. In the last selection policy, the latest event instances are chosen for pattern matching. In the above example, a detected complex event using the *last* selection policy for all event types might be $cplx = (A_1, B_3, C_5)$. Please note that we assume that a complex event is emitted whenever it is detected, i.e., the operator does not wait until the window closes to process events and emit complex events [CM94]. Waiting until the window closes to emit complex events might add a high latency to the detection of complex events.

Consumption Policy. In the above example, it is also unclear whether it is allowed to reuse the event instances while performing pattern matching or the event instances should be considered as consumed events (i.e., it is not allowed to reuse them again) whenever they have contributed to a complex event. The *consumption policy* [CM94; ZU99; CM10] defines whether the same event instance might be used in detecting multiple complex events. There are mainly two consumption policies: *consumed* and *zero*. In the *consumed* policy, it is *not allowed* to reuse the event instances in detecting other complex events. While in the *zero* policy, an event instance is *allowed* to contribute to multiple complex events. In the above example, let us assume that the selection policy is *first* for all event

types. Using the *consumed* consumption policy for all event types results in detecting only one complex event $cplx = (A_0, B_2, C_5)$. While using the *consumed* consumption policy for event types A and B and *zero* consumption policy for event type C results in detecting two complex events $cplx = (A_0, B_2, C_5)$ and $cplx^{'} = (A_1, B_3, C_5)$, where the event C_5 is reused in $cplx^{'}$. For more information on the selection and consumption policies, see [CM94; ZU99; CM10].

Please note that we do not assume a specific computational model or selection and consumption policy. Our proposed load shedding approaches support all aforementioned CEP computation models and selection and consumption policies. Moreover, as we mentioned above, there exist several event operators in CEP. However, we do not assume a specific event operator. In general, our load shedding approaches support all commonly used event operators.

2.2 Quality of Results

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

As 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}. In an operator, 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 2.1). 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 2.2).

$$FP_{\mathbb{Q}} = \sum_{q_i \in \mathbb{Q}} w_{q_i} * FP_{q_i} \tag{2.1}$$

$$FN_{\mathbb{Q}} = \sum_{q_i \in \mathbb{Q}} w_{q_i} * FN_{q_i} \tag{2.2}$$

As a result, for an operator, QoR is measured by the sum of the total number of false positives ($FP_{\mathbb{Q}}$) and the total number of false negatives ($FN_{\mathbb{Q}}$).

Recall that there might exist several instances of each event type within a window, where the selection and consumption policies 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 or dropping PMs 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 person A has commented on a post. Then, 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 or dropping PMs 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 2.1, Figure 2.3), let us consider window w_1 contains the following events: B_7, 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, 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., $cplx_o = (A_0, B_2, C_5)$– recall that in Example 1, we are matching events chronologically [CM94] (i.e., using the first selection policy and consumed consumption policy for all event types). Next, we first discuss the impact of event shedding on QoR. Then, we discuss the impact of PM shedding on QoR. Let us assume that due to event shedding, event B_2 is dropped from the window. In this case, the operator detects a new complex event $cplx_l^e$ from the events A_0, B_3, and C_5, i.e., $cplx_l^e = (A_0, B_3, C_5)$. Since the new complex event $cplx_l^e$ is not detected from the same event instances as the complex event $cplx_o$, in the strict QoR, complex event $cplx_l^e$ 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. Similarly, let us assume that the operator has

already processed events A_0, A_1, B_2, and B_3 in window w_1. Hence, the current open PMs in window w_1 after processing event B_3 are $\gamma_1 = (A_0, B_2)$ and $\gamma_2 = (A_1, B_3)$. Let us assume that due to PM shedding, PM γ_1 is dropped. In this case, after processing all events in the window w_1, the operator detects a new complex event $cplx_l^{PM}$ using PM γ_2 where it uses the events A_1, B_3, and C_5, i.e., $cplx_l^{PM} = (A_1, B_3, C_5)$. Since the new complex event $cplx_l^{PM}$ is not detected from the same event instances as the complex event $cplx_o$, in the strict QoR, complex event $cplx_l^{PM}$ is again 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 PM γ_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 beneficial 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 (a 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 events $cplx_l^e$ and $cplx_l^{PM}$ are considered equivalent to the complex event $cplx_o$. Hence, dropping event B_2 or PM γ_1 from window w_1 does not result in any false positive or negative in the case of relaxed QoR.

Please note that dropping events or PMs might result in false positives and negatives in the case of strict QoR. Moreover, it might result in false negatives when using relaxed QoR. However, when using relaxed QoR, dropping events might result in false positives only if the negation event operator is used. While dropping PMs does not result in false positives when using relaxed QoR, irrespective of the used event operator.

2.3 Problem Statement

In this section, we precisely define the problem solved in this thesis. Operators in a CEP operator graph are assigned latency bounds. A CEP operator might have limited resources where, in overload cases, it must perform load shedding to avoid violating the given latency bound (LB). Load shedding is performed by either dropping input events or partial matches. However, shedding load might degrade QoR, i.e., resulting in false positives and false negatives. Therefore, the load shedding must be performed in a way that has a minimum adverse impact on QoR.

As mentioned in Section 2.2, for an operator, QoR is measured by the the sum of the total number of false positives ($FP_{\mathbb{Q}}$) and the total number of false negatives ($FN_{\mathbb{Q}}$). For each operator in the CEP operator graph, the objective is to minimize the adverse impact on QoR, i.e., minimize ($FP_{\mathbb{Q}} + FN_{\mathbb{Q}}$), while dropping events and PMs such that the given latency bound LB is met. More formally, for each operator, the objective is defined as follows.

$$\begin{aligned} \textit{minimize} \quad & (FP_{\mathbb{Q}} + FN_{\mathbb{Q}}) \\ \text{s.t.} \quad & l_e \leq LB \quad \forall\ e \in S_{in} \end{aligned} \tag{2.3}$$

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.

Chapter 3

pSPICE: Partial Match Shedding

In this chapter, we present our first load shedding approach that is called pSPICE. pSPICE is a *white-box* load shedding approach that drops *partial matches*.

More specifically, pSPICE is an efficient and lightweight load shedding approach for CEP systems. In overload cases, pSPICE drops PMs to maintain a given latency bound, i.e., it sheds load on the PM granularity. As we mentioned in Chapter 1, Section 1.2, the event processing latency increases proportionally with the number of PMs in a CEP operator. Therefore, dropping PMs from the internal state of the operator reduces the event processing latency and increases the operator throughput. Hence, it enables the operator to maintain a given latency bound in case of input event overload. Of course, shedding PMs might influence QoR. Therefore, it is crucial to drop PMs that have a low adverse impact on QoR. To reduce the negative shedding impact on QoR, pSPICE drops PMs that have the lowest importance. Please recall that dropping PMs has an advantage over dropping events where if the relaxed QoR is used, dropping PMs does not result in false positives. That might be important for applications that cannot tolerate false positives.

There are three main challenges to drop partial matches in CEP: 1) determining when and how many PMs to drop for an incoming input event rate, 2) determining which PMs to drop, and 3) performing the load shedding in a lightweight manner so as not to burden an already overloaded operator. To drop PMs, we associate each PM with a utility value that indicates the importance of the PM where a higher utility value means higher importance. We derive the utility of a PM from its probability to complete and become a complex event (called partial match completion probability) and from its estimated remaining processing time.

Our main contributions in this chapter are as follows:

- We propose a white-box load shedding strategy, called pSPICE, that uses the Markov chain and Markov reward process to predict the utility of PMs in windows. The utility of a PM depends on the completion probability of the PM and on its remaining processing time.

- We develop an approach that enables pSPICE to perform the load shedding in an efficient and lightweight manner.
- We provide an algorithm that decides when and estimates how many PMs to drop from an operator to maintain the given latency bound.
- We provide extensive evaluations on three real-world datasets and several representative queries to show that pSPICE reduces the adverse impact of load shedding on QoR considerably more than state-of-the-art solutions.

The rest of the chapter is structured as follows. Section 3.1 presents the used system model. In Section 3.2, we explain in detail how different components of pSPICE interact, how the utility of PMs are defined and predicted in pSPICE, and how load shedding is performed. Section 3.3 presents the obtained evaluation results. Finally, we conclude this chapter in Section 3.4.

3.1 System Model

We rely on a similar system model as in Chapter 2, Section 2.1. We assume a window-based CEP system that consists of a one or more operators where an operator might detect one or more patterns $\mathbb{Q}$ (i.e., multi-query). Patterns might have different importances where each pattern has a corresponding weight $w_{q_i} \in \mathbb{W}_{\mathbb{Q}}$ (given by the domain expert) reflecting the pattern's importance. A pattern $q_i \in \mathbb{Q}$ is modeled as a finite state machine. For a pattern $q_i \in \mathbb{Q}$, we define a set of states $\mathbb{S}_{q_i} = \{s_0, s_1, ..., s_m\}$ as the set of all possible states that the pattern q_i may have, including the initial state (s_0). Assuming that the state s_m represents the final state of pattern q_i, a PM γ of pattern q_i (i.e., $\gamma \subset q_i$) completes and becomes a complex event if an instance of the state machine of pattern q_i transits to the state s_m.

In this chapter, we assume a white-box CEP operator where the operator reveals information about PMs and their progress (i.e., states) when processing primitive events within windows. As we mentioned in Chapter 2, Section 2.1, in CEP systems, there exist several computational models other than the finite state machine that are used to detect patterns. Please note that pSPICE is not restricted to a specific computational model. Later in this chapter, we show how pSPICE supports other computational models.

3.2 pSPICE

In this section, we first present the architecture of pSPICE, our load shedding strategy. Next, we introduce the notion of utility of PMs followed by a description of our approach to determine these utilities using the Markov chain and Markov reward process [How12; How13]. Then, we discuss how to detect overload and compute the amount of overflowing PMs that must be dropped by the load shedder. After that, we present the load shedding

algorithm that efficiently drops PMs with the lowest utility values. Finally, we discuss how pSPICE supports other computational models. Please note that in Section 2.2, we mentioned two types of QoR, namely, strict and relaxed QoR. Our shedding approaches in this thesis are designed depending on the strict QoR. However, we show evaluation results with both strict QoR and relaxed QoR.

3.2.1 The pSPICE Architecture

The architecture of pSPICE is depicted in Figure 3.1. The figure shows an operator which is modified by adding the following components to enable load shedding: overload detector, load shedder (LS), and model builder.

The incoming windows of events forwarded by the merger-splitter component (cf. Chapter 2, Section 2.1) are queued in the input queue of the operator. To prevent violating the defined latency bound (LB), the overload detector checks the estimated latency for each input event. In the scenario where LB might be violated, the overload detector calls the LS to drop a certain number of PMs, denoted by ρ.

The model builder receives observations from the operator about the progress of PMs. After receiving a certain number of observations, the model builder builds the model, where it predicts the utility of PMs using the Markov chain and Markov reward process. The model builder might be heavy-weight. However, it is not a time-critical task and it does not need to run frequently.

LS drops ρ PMs every time it is called by the overload detector, where ρ is determined by the overload detector. The LS depends on utility values predicted by the model builder to select those PMs for dropping. Both the LS and overload detector are time-critical tasks, which directly affect the CEP system performance and hence must be lightweight and efficient. As we will see later, both of these components have very low overhead in pSPICE.

3.2.2 Utility of Partial Matches

pSPICE drops partial matches with the lowest utility. The question is– what defines the utility of a PM? The utility of a PM is defined by its impact on QoR, i.e., the number of false negatives and positives. A PM that has a low adverse impact on QoR has a low utility value, while a PM that has a high adverse impact on QoR has a high utility value. Hence, to minimize the dropping impact on QoR, we must find a way to assign low utility values to those PMs that are less important than other PMs. We assign utilities to PMs depending on three factors: 1) the probability of a PM to complete and become a complex event (i.e., the completion probability), 2) the estimated processing time that a PM still needs, and 3) the weight of the pattern.

The completion probability of a PM represents the probability of the PM to become a complex event. The existence of a complex event depends on whether its underlying PM

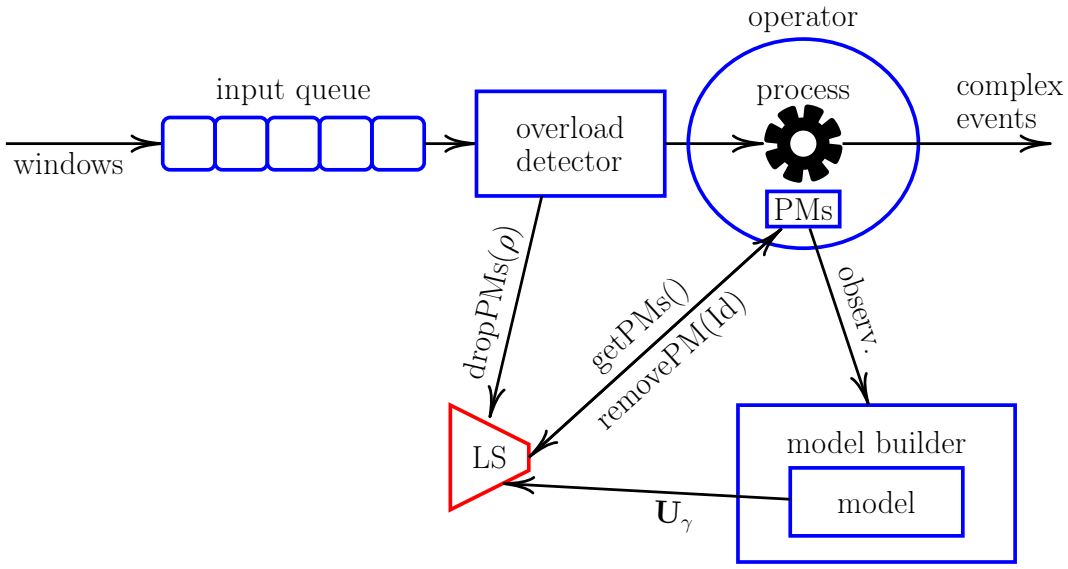


Figure 3.1: The pSPICE Architecture.

will complete or not. If a PM completes, a complex event is detected. On the other hand, if a PM does not complete, a complex event is not detected. Hence, the completion probability of a PM is an important indicator of the utility of the PM as dropping PMs that anyway will not complete might imply no degradation in QoR. P_γ represents the completion probability of the PM γ. The higher is the completion probability P_γ of the PM γ, the higher should be its utility. This means that the utility of a PM is proportional to its completion probability.

The utility of a partial match γ is also influenced by its remaining processing time (denoted by τ_γ). A PM that still has a high remaining processing time (we will use only processing time hereafter) should have lower utility than a PM that has a lower processing time. The reason for this is that a PM with low processing time consumes less processing time from the operator, i.e., giving the operator more time to process other PMs. Hence, it decreases the need to drop PMs from the operator's internal state, which in turn decreases the number of false negatives and positives. This means that the utility of a PM is inversely proportional to its processing time (τ_γ).

For example, let us assume that an operator has two partial matches γ_1 and γ_2 in two windows w_1 and w_2, respectively. Suppose that $P_{\gamma_1} = P_{\gamma_2}$ but $\tau_{\gamma_1} < \tau_{\gamma_2}$. In this case, the importance of γ_1 should be higher than the importance of γ_2 since γ_1 has the same completion probability as γ_2 but it imposes lower processing time on the operator. In another case where $P_{\gamma_1} < P_{\gamma_2}$ but $\tau_{\gamma_1} < \tau_{\gamma_2}$, we need to assign a higher utility to the PM that results in lesser degradation in QoR. Therefore, we use the proportion of the completion probability P_γ to the processing time τ_γ, i.e., $\frac{P_\gamma}{\tau_\gamma}$, as a utility value for the partial match.

Finally, as we mentioned above, in an operator with multiple patterns (i.e., multi-query

operator), each pattern might have different weight $w_{q_i} \in \mathbb{W}_{\mathbb{Q}}$, i.e., different importance. Therefore, when assigning utilities to PMs, we must also take the patterns' weights into consideration. To consider the pattern's weights, we increase the utility value of a PM $\gamma \subset q_i$ proportionally to its pattern's weight w_{q_i}.

For a PM $\gamma \subset q_i$, to incorporate the completion probability P_γ of the PM γ, its processing time τ_γ, and its pattern's weight w_{q_i} in deriving the utility (denoted by U_γ) of the PM γ, we represent the utility of PM γ as follows:

$$U_\gamma = w_{q_i}.\frac{P_\gamma}{\tau_\gamma} \tag{3.1}$$

3.2.3 Utility Prediction

Since the utility of a PM depends on its completion probability and processing time, in this section, we explain the manner in which we predict them using the Markov chain and Markov reward process [How12; How13].

3.2.3.1 Completion probability Prediction

In a certain position in a window w, the completion probability P_γ of a PM $\gamma \subset q_i$, i.e., the probability of γ to complete the pattern q_i and to become a complex event, depends on two factors. 1) on **the current state** of the PM γ (denoted by S_γ), where $S_\gamma \in \mathbb{S}_{q_i}$, and 2) on **the number of remaining events** in the window w (denoted by R_w). Therefore, we write P_γ as a function of S_γ and R_w as follows:

$$P_\gamma = f(S_\gamma, R_w) \tag{3.2}$$

$S_\gamma = s_0$ means that PM γ is at the initial state, while $S_\gamma = s_m$, where $m = (|\mathbb{S}_{q_i}| - 1)$, means that γ has completed and become a complex event. $R_w \in [1, ws]$, where ws represents the expected window size. If a partial match γ has a state S_γ that is close to the final state and R_w is high, the probability for γ to complete and become a complex event might be high. That is because PM γ needs only fewer state transitions to reach the final state and the window w still has a high number of events that can be used to match the pattern q_i and to complete γ. On the other hand, the completion probability might be low for a partial match γ that has a state S_γ which is close to the initial state and R_w is low. That is because PM γ still needs many state transitions to reach the final state and the window w only has a small number of events that can be used to match the pattern q_i and to complete PM γ.

Since a pattern in CEP systems can be represented as a state machine, as we mentioned above, in this work, we model the pattern matching process as a Markov chain to predict the completion probability P_γ of a partial match γ of a pattern. To clarify this, let us introduce the following simple example. Let us assume that an operator matches a pattern $q_i = seq(A; B; C)$. This pattern can be represented as a state machine as depicted

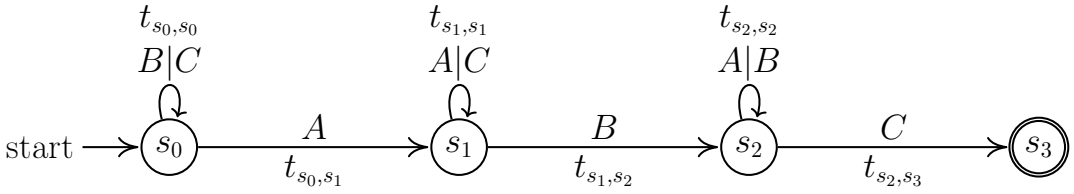


Figure 3.2: State machine example.

in Figure 3.2, where it has four states, including the initial state, i.e., $\mathbb{S}_{q_i} = \{s_0, s_1, s_2, s_3\}$. The state machine transitions from one state to other states depending on the input symbols (events), while the Markov chain probabilistically transitions from one state to other states using a *transition matrix*. In the above example, if we assume that the input event stream has only three event types (A, B, and C) and instances of these events are coming randomly with a uniform distribution, then the probability to transition from any state to the next state is 1/3. While the probability to stay in the same state is 2/3.

Therefore, the transition matrix can be used to predict the probability of the state machine to transition from any state to other states and hence to predict the probability of the state machine to transition from a certain state s_i to the final state s_m after processing R_w input symbols. Since a PM is represented as an instance of the state machine, the completion probability of a PM (i.e., the probability of the state machine to reach the final state) at a certain state S_γ, given that R_w events are left in the window w, i.e., $P_\gamma = f(S_\gamma, R_w)$, can be computed using the transition matrix. Since the input event stream might follow any distribution, not only uniform distribution, we should learn the transition matrix by gathering statistics about the state transitions of PMs as we describe next.

Statistic gathering & Transition matrix: For each pattern $q_i \in \mathbb{Q}$, the model builder builds a transition matrix T_{q_i} from the statistics gathered during run-time by monitoring the internal state of the operator. The statistics contain information about the progress of PMs of pattern q_i when processing the input events within windows. For each partial match $\gamma \subset q_i$, the operator reports, when processing an input event e within a window, whether PM γ progressed or not, i.e., the state of γ changed or not by processing the event e with PM γ. The operator forms an $Observation\langle q_i, s, s' \rangle$, where s represents the current state of PM γ and s' the new state of PM γ after processing one event in the window.

After gathering statistics from η observations for pattern q_i, the model builder transfers these statistics to the transition matrix T_{q_i}. T_{q_i} describes the *transition probability* between the states of the Markov chain when processing *one event* in a window.

Completion probability: As mentioned above, the transition matrix gives the probability to transition from one state to another state and can be used to predict the partial match completion probability P_γ. Figure 3.3 shows the transition matrix for the state machine depicted in Figure 3.2. Since we are only interested to know

$$
\begin{array}{c}
\text{next state} \\
\begin{array}{c} \\ \text{current state} \end{array}
\begin{array}{c}
\begin{array}{cccc} s_0 & s_1 & s_2 & s_3 \end{array} \\
\begin{array}{c} s_0 \\ s_1 \\ s_2 \\ s_3 \end{array}
\left[\begin{array}{cccc}
p_{00} & p_{01} & p_{02} & p_{03} \\
p_{10} & p_{11} & p_{12} & p_{13} \\
p_{20} & p_{21} & p_{22} & p_{23} \\
p_{30} & p_{31} & p_{32} & p_{33}
\end{array}\right]
\end{array}
\end{array}
$$

Figure 3.3: Transition matrix T_{q_i} for the state machine in Figure 3.2.

whether a partial match will complete or not, we need only to focus on the last column in the transition matrix, surrounded by a red box in Figure 3.3. This column gives the probability to move from any state $S_\gamma \in \mathbb{S}_{q_i}$ to the final state, i.e., the probability to complete the partial match.

The transition matrix contains the transition probability, given that there is only one event left in a window. Therefore, to get the transition probability given that R_w events are still left in a window w, we must raise the transition matrix T_{q_i} to the power R_w [How12]. This way, the completion probability of a partial match γ at a state S_γ, given that R_w events are left in a window w, is computed as follows:

$$P_\gamma = f(S_\gamma, R_w) = T_{q_i}^{R_w}(j, m) \tag{3.3}$$

where $S_\gamma = s_j \in \mathbb{S}_{q_i}$ and $m = (|\mathbb{S}_{q_i}| - 1)$, $m = 3$ in the above figure. For example, in Figure 3.3, the completion probability of a partial match $\gamma \subset q_i$ at the state s_2 given that only one event ($R_w = 1$) is left in a window w is computed as follows: $P_\gamma = f(s_2, 1) = T_{q_i}^1(2, 3) = p_{23}$. To get the completion probability of a partial match given any number of events are left in a window, we need to compute the transition matrix $T_{q_i}^{R_w}$ for all possible values of $R_w \in [1, ws]$. However, the window size ws might be too large which might impose a high memory cost during the calculation of the transition matrices. Therefore, we calculate the transition matrix only for every bs (i.e., bin size) events, i.e., $T_{q_i}^{bs}$, $T_{q_i}^{2.bs}$, ..., $T_{q_i}^{ws}$. To get the completion probability of a PM in case $R_w \in [(j-1).bs,\ j.bs]$, where $j = 1, 2, ..., \frac{ws}{bs}$, we use linear interpolation. For ease of presentation, we assume that $bs = 1$, if not otherwise stated.

3.2.3.2 Processing Time Prediction

After predicting the completion probability of PMs, now, we describe how to predict the processing time of PMs using the Markov reward process [How13], where we model the processing time of a PM as the reward value. Given a partial match $\gamma \subset q_i$ at a state S_γ, we define the time that is needed to match an event in a window w with γ as $t_{s,s'}$, where $s = S_\gamma$ and $s' \in \mathbb{S}_{q_i}$. Hence, $t_{s,s'}$ represents the processing time that is needed for the state machine of q_i to transition from state s to state s'. For example, in Figure 3.2,

the processing time to transition from s_1 to s_2 is represented by the value t_{s_1,s_2}. We consider $t_{s,s'}$ as a reward value to move from state s to state s' in the state machine of pattern q_i.

Therefore, to calculate the processing time of a PM, we clearly need something more than using the Markov chain which is used to compute the completion probability of a PM. As a result, we upgrade our Markov chain to a Markov reward process, where we additionally define the reward function $R_{q_i}(s, s')$ as the expected processing time needed to transition from state s to state s'. Solving the Markov reward process gives us the expected reward for each state in the state machine, given that there are still R_w events left in a window w. Since we represent the processing time $t_{s,s'}$ as a reward, the reward of a state represents the estimated processing time of a PM τ_γ, given that there are still R_w events lefts in window w.

We incorporate the processing time $t_{s,s'}$ in statistics gathering and extend the above observation as follows: $Observation\langle q_i, s, s', t_{s,s'}\rangle$. After gathering statistics from η observations for pattern q_i, the model builder constructs the reward function (i.e., $R_{q_i}(s, s')$) which is calculated as the average value for all observed values of the processing time $t_{s,s'}$. After that, the model builder predicts the processing time of PMs by solving the Markov reward process as we explain next.

Processing time: To predict the processing time of a partial match $\gamma \subset q_i$ in a state S_γ, the model builder must solve the Markov reward process. A well-known algorithm called *value iteration* [How13] can be used to solve the Markov reward process. The algorithm iteratively calculates the expected reward (processing time τ_γ in this case) at every state in the state machine using the transition matrix T_{q_i} and the reward function R_{q_i}. Then, it reuses the calculated reward values in future iterations. Here, an iteration j represents the number of remaining events (i.e., R_w) in a window w, i.e., $j = R_w$. The value iteration algorithm uses the Bellman equation [Bel57] to predict the remaining processing time τ_γ of a partial match $\gamma \subset q_i$ at state S_γ given that there are still R_w events left in the window w.

Similar to the completion probability, we run the value iteration algorithm to get the processing time τ_γ of a partial match γ for all expected remaining R_w number of events in a window w. To avoid the memory overhead in the case of too large window size ws, again, we keep the value iteration results only for every bs events. For the intermediate values, we use linear interpolation.

3.2.3.3 Utility calculation

After describing how to predict the completion probability and the processing time of a PM, now, we can derive the utility of PMs for each pattern $q_i \in \mathbb{Q}$ using Equation 3.1.

Since the completion probabilities and processing times of PMs have different units and scales, using Equation 3.1 directly on these values, may result in unexpected behavior, where a high processing time may overcome the completion probability and eliminate

its importance in calculating the utility of PMs. Therefore, before using Equation 3.1, we bring the completion probabilities and processing times to the same scale and then apply Equation 3.1 to get utilities of PMs.

To efficiently retrieve the utilities by the LS, we store the utility of PMs at any given state and for any number of remaining events in a window in a table called UT_{q_i}, where each pattern q_i has its corresponding utility table. UT_{q_i} has $(m\ X\ \frac{ws}{bs})$ dimensions, where $m = (|\mathbb{S}_{q_i}| - 1)$ and each cell $UT_{q_i}(j,k)$ represents the utility of a PM at state s_j given that there are still k events left in the window, assuming $bs = 1$. So the utility of a PM $\gamma \subset q_i$ is calculated as follows: $U_\gamma = f(S_\gamma, R_w) = UT_{q_i}(j,k)$, where $s_j = S_\gamma$ and $k = R_w$. Getting the utility of a PM from UT has only O(1) time complexity which is a great factor in minimizing the overhead of the LS.

3.2.4 Model Retraining

The event distribution in the input event stream and/or the content of input events may change over time and hence our model might become inaccurate and adversely impact QoR. To avoid this, we must retrain the model to capture those changes. The question is– how do we know that those changes happened and the model must be retrained? We depend on the transition matrix to answer this question.

The transition matrix, as we know, contains the probabilities to transition from any state to other states in the state machine, where the transition matrix is constructed depending on the distribution of the input event stream and on the content of events. If there is a change in the distribution of the input event stream and/or in the content of events, the probability values in the transition matrix will change. Therefore, the transition matrix can be used as an indicator of those changes and to trigger model retraining. Hence, we propose to periodically build a new transition matrix from the gathered statistics from the operator and compare the new transition matrix with the transition matrix that is used in the model by using an error measurement, e.g, mean squared error. If the deviation between the two matrices is higher than a threshold, the model builder must rebuild the model. Please note that building a new transition matrix is lightweight since we just need to transfer the gathered statistics about the state transitions to probability values. Moreover, we don't need to calculate new transition matrices for all expected remaining number of events in a window to check for the need to retrain the model.

3.2.5 Detecting and Determining Overload

The goal of pSPICE is to avoid violating a defined latency bound (LB). A high queuing latency of the incoming input events in the operator input queue indicates an overload on the operator and hence some partial matches must be dropped from the operator's internal state to avoid violating LB. Algorithm 1 specifies the functionality of the overload detector.

Algorithm 1 Detecting and Determining Overload.

```
1:  detectOverload (event e) begin
2:  l_q = currentTime() − e.arrivalTime()
3:  l_p = f(n_γ), l_s = g(n_γ)                    // n_γ: Current number of PMs.
4:  l_e = l_q + l_p
5:  if l_e + l_s > LB then                        // LB might be violated => drop PMs.
6:      l'_p = LB − l_q − l_s
7:      n'_γ = f^{-1}(l'_p)
8:      ρ = n_γ − n'_γ
9:      LS.dropPMs(ρ)                             // Call LS to drop ρ PMs.
10: end function
```

Detecting overload: The overload detector continuously gets the primitive events from the input queue of the operator, where for each event, it checks whether LB might be violated. In the case where LB might be violated, the overload detector calls the load shedder to drop a certain number of PMs to reduce the overhead on the operator and maintain LB. The violation of LB depends on the estimated event latency (l_e) and load shedding latency (denoted by l_s), where LB would be violated if the following inequality holds:

$$l_e + l_s > LB \tag{3.4}$$

Recall that the estimated event latency l_e represents the time between the insertion of the event e in the operator's input queue and the time when the event e is processed by the operator in all currently opened windows, since an event may belong to several windows in case windows overlap. The load shedding latency l_s represents the time needed by the LS to drop the required number of partial matches.

The estimated event latency l_e of an event e is the sum of the event queuing latency (denoted by l_q) and the estimated event processing latency (denoted by l_p): $l_e = l_q + l_p$. The event queuing latency l_q is the time between the insertion of the event e in the operator's input queue and the time when the operator gets the event e from its input queue to process it (cf. Algorithm 1, line 2). While the estimated event processing latency l_p represents the time an event e needs to be processed by the operator in all currently opened windows. l_p depends on the current number of partial matches (denoted by n_γ) in the operator since the event e needs to be matched with all current partial matches in the operator. The higher is the value of n_γ, the higher is l_p. Therefore, we represent l_p as a function, called event processing latency function, of the current number of partial matches n_γ in the operator: $l_p = f(n_\gamma)$, i.e., $f : n_\gamma \rightarrow l_p$.

Therefore, for each event e, the overload detector calls the event processing latency

function $f(n_\gamma)$ that gives the estimated event processing latency l_p depending on the current number of partial matches in the operator (cf. Algorithm 1, line 3). Using l_p and l_q, the overload detector can now compute the estimated event latency l_e (cf. Algorithm 1, line 4). To build the function $f(n_\gamma)$, during run-time, we gather statistics from the operator on the event processing latency l_p for different numbers of partial matches n_γ. Then, we apply several regression models on these statistics to get the function $f(n_\gamma)$, where we use a regression model that results in a lower error.

We consider the load shedding latency l_s in the inequality (3.4) since during load shedding no events are processed and hence the event queuing latency is increased by the time needed to drop PMs, i.e., by the load shedding latency l_s. Similar to the estimated event processing latency, the load shedding latency l_s also depends on the current number of PMs n_γ. That is because the load shedder must sort all current PMs in the operator to find those PMs that have the lowest utility values (we will show this later). Therefore, we also represent l_s as a function of n_γ: $l_s = g(n_\gamma)$ (cf. Algorithm 1, line 3). Similarly, to build the function $g(n_\gamma)$, during run-time, we gather statistics from the operator on the load shedding latency l_s for different numbers of PMs n_γ. Then, we apply several regression models on these statistics to get the function $g(n_\gamma)$, where we use a regression model that results in a lower error.

Determining overload amount: As we explained above, if the inequality (3.4) holds, the overload detector calls the LS to drop PMs to avoid violating LB (cf. Algorithm 1, lines 5-9). The question is– how many PMs must the LS drop? To answer this question, we need to understand which latency values in the inequality (3.4) can be controlled. We cannot reduce the event queuing latency l_q and the load shedding latency l_s but we can reduce the event processing latency l_p by dropping some PMs. Therefore, we represent the new event processing latency as l'_p such that the following condition holds.

$$l'_p + l_q + l_s = LB. \tag{3.5}$$

From the above condition, $l'_p = LB - l_q - l_s$. Therefore, we have to ensure the new processing latency l'_p by dropping a certain number of PMs (denoted by ρ).

To compute ρ, we should find the number of PMs (denoted by n'_γ) that impose a latency of l'_p on the operator when processing an event. Hence, n'_γ is a function of l'_p. This function is the inverse function f^{-1} of the event processing latency function $f(n_\gamma)$, where $f^{-1} : l'_p \rightarrow n'_\gamma$. From the inverse function f^{-1}, we can compute the number of PMs n'_γ. Keeping only n'_γ PMs in the operator's internal state ensures that the operator needs only l'_p time to process an event and hence it maintains LB. Therefore, the number of PMs to drop $\rho = n_\gamma - n'_\gamma$. For each input event, the overload detector calls the LS to drop ρ partial matches whenever the inequality (3.4) holds (cf. Algorithm 1, line 9).

Please note that the inequality (3.4) ensures to keep the event latency l_e less than or equal to LB. However, in case of a sudden increase in the input event rate or inaccuracy in the functions that predict l_p and l_s, there might be a risk of violating LB. Therefore,

in latency critical applications where LB is a hard bound, we propose to add a safety buffer (denoted by b_s) to the inequality (3.4) as follows:

$$l_e + l_s + b_s > LB \tag{3.6}$$

3.2.6 Load Shedding

In this section, we discuss the functionality of the LS component that is called by the overload detector to drop PMs. The LS drops PMs with the lowest utility values, where the utility of PMs is learned and stored in UT as we explained in Section 3.2.3. Algorithm 2 specifies the functionality of the LS.

Whenever the LS is called by the overload detector to drop ρ PMs, it needs to know the current ρ PMs in the operator that have the lowest utility values. To get the utility of PMs, the LS simply uses the utility tables given by the model builder. For a PM $\gamma \subset q_i$ in a window w, the LS obtains the utility of PM γ, i.e., U_γ, by a simple lookup in the utility table UT_{q_i}. $U_\gamma = UT_{q_i}(j,k)$, where $S_\gamma = s_j \in \mathbb{S}_{q_i}$, and $k = R_w$, i.e., the expected number of events left in the window w (cf. Algorithm 2, lines 2-4). Therefore, the time complexity to get the utility of a PM is O(1), and hence to get the utility for all current PMs in the operator is O(n_γ), where n_γ represents the number of current open PMs in the operator. To find the ρ PMs with the lowest utility values among all PMs, the LS should sort the PMs using their utility values, where a good sorting algorithm (e.g., merge sort [Knu98; Ski08]) can achieve O($n_\gamma\ log_2(n_\gamma)$) average time complexity (cf. Algorithm 2, line 5). After sorting PMs, the LS drops the first ρ PMs which have the lowest utilities, where the LS iterates over the sorted PMs and asks the operator to remove those PMs from its internal state (cf. Algorithm 2, lines 6-10). This has a time complexity of O(ρ). Hence, the overall time complexity for the load shedding is O($n_\gamma + n_\gamma\ log_2(n_\gamma) + \rho$). As we will show in Section 3.3, the overhead of our LS is extremely low.

3.2.7 Supporting CEP Computational Models

So far, we have focused on using a finite state machine [May+17; Agr+08; CM10; RLR16; WDR06] as a computational model to detect patterns. However, as we mentioned in Chapter 2, Section 2.1, there exist several other computational models such as tree-based models [CM94; Cha+94; MM09], string-based models [Sad+04], and Petri Nets-based models [GD94]. 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 U_γ to a PM γ, pSPICE depends on two features: 1) the current state S_γ of the PM γ, and 2) the number of remaining events R_w in the window. The way we can get the current state S_γ of the PM γ depends on the used computational model, however, the number of remaining events R_w in the window

Algorithm 2 Load Shedding.

```
1:  dropPMs (ρ) begin
    // get utilities of PMs and sort them.
2:  for each γ in operator.getPMs() do
3:      U_γ = getUtility(q_i, S_γ, R_w), where γ ⊂ q_i
4:      pmArray.insert(γ)
5:  sortByUtility(pmArray)
    // drop ρ partial matches.
6:  for index = 0 → ρ do
7:      if index >= pmArray.size() then          // No more PMs to drop!
8:          return
9:      γ = pmArray(index)
10:     operator.removePM(γ)
11: end function
```

is *independent* of the computational model. Therefore, to show that pSPICE supports other computational models, we must show that pSPICE is able to get PM states in these computational models, similar to the finite state machine model. To do that, let us first recall the definition of 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 [CM94; Luc01]. Pattern q has a set of states $\mathbb{S}_q = \{s_0, s_1, ..., s_m\}$, where we assign a distinct state to each event type in pattern q, and s_0 represents the initial state of pattern q. For pattern q, a PM γ of pattern q represents an incomplete matching instance of pattern q, denoted by $\gamma \subset q$. Moreover, PM γ is assigned a state $s_i \in \mathbb{S}_q$ corresponding to the last event type that has been matched in PM γ. For example, we may assume the following set of states for pattern $q = seq(A; B; C)$: $\mathbb{S}_q = \{s_0, s_1, s_2, s_3\}$. Here s_0 represents the initial state. Moreover, we 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 $\gamma \subset q$ starts at state s_0 where it completes and becomes a complex event if it reaches state s_3. 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, pSPICE can support other computational models without any remarkable additional complexity.

3.3 Performance Evaluations

In this section, we show the performance of pSPICE by evaluating it with three real world datasets and several representative queries. We assume that a CEP operator graph that consists of a single operator and the operator my match one or more queries, as shown below.

3.3.1 Experimental Setup

Evaluation Platform. We run our evaluation on a machine that is equipped with 8 CPU cores (Intel 1.6 GHz) and the 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 that the resource limitation can be any number of threads/cores, and the behavior of pSPICE does not depend on a specific limitation. We implemented pSPICE by extending a prototype Java-based CEP framework, which is implemented at the department of distributed systems at the University of Stuttgart.

Baseline. We also implemented two other load shedding strategies to use as baselines. 1) We implemented a random partial match dropper (denoted by PM-BL) that uses Bernoulli distribution [Ber] to drop PMs. 2) We also implemented a load shedding strategy (denoted by E-BL) similar to the one proposed in [HBN14]. In addition, it captures the notion of weighted sampling techniques in stream processing [Tat+03]. E-BL assigns utility values to the events in a window depending on the repetition of those 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, E-BL decides the number 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. E-BL, as in [HBN14], does not consider the order of events in a pattern and in the input event stream.

Datasets. We use three real-world datasets. 1) A stock quote stream from the New York Stock Exchange (NYSE), which contains real intra-day quotes of different stocks from NYSE collected over two months from Google Finance [Goo]. 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. 2) A position data stream from a real-time locating system (denoted by RTLS) in a soccer game [MZJ13]. Players, balls, and referees (called objects) are equipped with sensors that generate events. Events contain information about those objects, such as 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. 3) Public bus traffic (denoted by *PLBT*) from a real transportation system in Dublin city [Zac+15]. It contains events from 911 buses, where each event has information about those buses, e.g., locations, stops, delays, etc.

Queries. We apply four queries (Q_1, Q_2, Q_3, and Q_4) that cover an important set of operators in CEP as shown in Table 3.1: sequence operator, sequence operator with repetition (which also contains Kleene closure), sequence with any operator, and any operator [ZDI14; CM94; CM10; WDR06; MM09]. Please note that, as mentioned in Section 2.1, we do not consider a certain selection or consumption policy. However, evaluating pSPICE with all possible combinations of selection and consumption policies is time-consuming and might be practically infeasible. Therefore, in this thesis, we will use the following important selection and consumption policies. 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 of the events in all queries. Moreover, the queries use both **time-based** and **count-based** sliding window strategies with **different predicates**.

In Table 3.1, we use ws to refer to the window size/length. For stock queries (Q_1 and Q_2), C_i represents the stock quote of company i. Q_1 (sequence operator) detects a complex event when rising or falling stock quotes of 10 certain stock symbols, by a given percentage, are detected within ws events in a certain sequence. Q_2 (sequence operator with repetition) 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 events in a certain sequence. Q_3 (sequence with any operator) uses the RTLS dataset and it detects a complex event when any n defenders of a team (defined as $\texttt{D}_\texttt{i}$) defend against a striker (defined as $\texttt{S}$) 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. Q_4 (any operator) uses the PLBT dataset. It detects a complex event when any n buses (defined as $\texttt{B}_\texttt{i}$) within a window of size ws events get delayed at the same stop.

3.3.2 Experimental Results

In this section, we evaluate the performance of pSPICE using relaxed QoR. In the next chapters, we also show the impact of pSPICE when using strict QoR. Please recall that pSPICE does not result in false positives when using relaxed QoR. Therefore, in this section, we evaluate the performance of pSPICE only w.r.t. false negatives. We first compare its performance, w.r.t. number of false negatives, with PM-BL and E-BL. Then, we show the importance of using the processing time of a PM in calculating its utility. Finally, we present the overhead of pSPICE.

If not stated otherwise, we use the following settings. For Q_1 and Q_2, we use a *count-based* sliding window. For both queries, we use a *logical* predicate (i.e., pattern-based predicate) where a new window is opened for each incoming event of the leading stock symbols. We choose 4 important companies as leading stock companies. Q_3 uses a *time-based* sliding window. Again, we use a *logical* predicate for Q_3, where a new window is opened for each incoming striker event (S). For Q_4, we use a *count-based* sliding

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 events
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 events
Soccer query	
Q_3	**pattern seq**$(S;$ **any**$(n, D_1, D_2, .., D_m))$ **where** S *possesses ball* **and** $distance(S, D_i) \leq x$ *meters* , $i = 1..m$ and m is the number of players in a team **within** ws seconds
Bus query	
Q_4	**pattern any**$(n, B_1, B_2, .., B_m)$ **where** $B_i.delay > x$ *seconds* **and** *all* B_i *have the same stop* , $i = 1..m$ and m is the number of buses **within** ws events

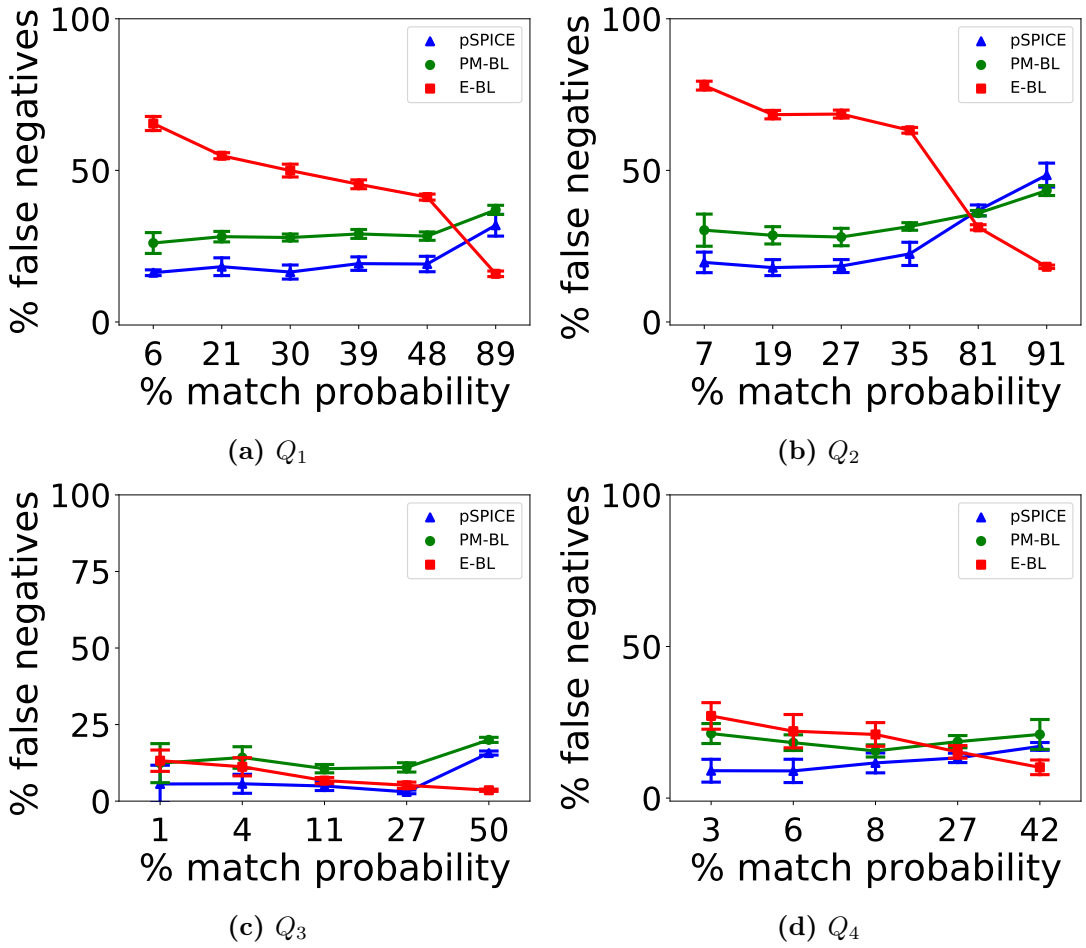
Table 3.1: Queries.

window and a *count-based* predicate, where a new window is opened every 500 events, i.e., slide size is 500 events. We stream events to the operator from datasets that are stored in files where we first stream events at event input rates that are less or equal to the maximum operator throughput 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 execute several runs for each experiment and show the mean value and standard deviation.

3.3.2.1 Impact on QoR and the given latency bound.

Now, we show the performance of pSPICE w.r.t. its impact on QoR (i.e., number of false negatives) and maintaining the given latency bound. Two factors influence the performance of pSPICE: 1) match probability, and 2) input event rate. Match probability represents the percentage of PMs that complete and become complex events out of all PMs. It is computed from the ground-truth by dividing the total number of complex events by the total number of PMs. We can control the match probability by varying the pattern size and/or the window size.

Impact of match probability. To evaluate the performance of pSPICE with different match probabilities, we run experiments with Q_1, Q_2, Q_3, and Q_4. For Q_1 and Q_2, we use a variable window size to control the match probability since Q_1 and Q_2 have a



(a) Q_1 (b) Q_2

(c) Q_3 (d) Q_4

Figure 3.4: Impact of match probability.

fixed pattern size. Higher is the window size, higher is the match probability. We use the following window sizes for Q_1: ws = 3.5K, 4.5K, 5K, 5.5K, 6K, 10K events. For Q_2, the used window sizes are: ws = 6K, 7K, 7.5K, 8K, 12K, 14K events. For Q_3 and Q_4, we use a fixed window size but a variable pattern size. For Q_3, we use a window size ws of 15 seconds and the following pattern sizes (i.e., number of defenders): n= 2, 3, 4, 5, 6. The window size ws for Q_4 is 8K events and we use the following pattern sizes (i.e., number of buses): n = 3, 4, 7, 8, 10. Moreover, we stream all datasets to the operator with an event input rate that is higher than the maximum operator throughput by 20% (i.e., event rate= 120% of the maximum operator throughput).

Figure 3.4 shows results for all queries, where the x-axis represents the match probability and the y-axis represents the percentage of false negatives. A low match probability means that most of the PMs don't complete, and hence dropping those PMs, that will not complete, decreases the dropping impact on QoR. On the other hand, a high match probability means that most of the PMs complete and become complex events, and hence dropping any PM may result in a false negative. This is observed in Figure 3.4 for all queries (Q_1, Q_2, Q_3, Q_4). Figure 3.4a depicts the results for Q_1, where it shows that

the percentage of false negatives produced by pSPICE increases with increasing match probability. It increases from 16% to 32% when the match probability increases from 6% to 89%, respectively. We observed similar behavior for PM-BL, where the percentage of false negatives increases from 26% to 37% when the match probability increases from 6% to 89%, respectively. As we observe from the figure, a high match probability degrades the performance of pSPICE since dropping any PM might result in a false negative as all PMs have a similar completion probability. In this experiment, pSPICE reduces the percentage of false negatives by up to 70% compared to PM-BL. Please note that a high rate of PM drop is because the operator load doesn't come only from processing PMs but also from managing windows and events and checking whether an event opens a partial match.

The performance of E-BL is bad when the match probability is low and it becomes better with a higher match probability as shown in Figure 3.4a. This is because a low match probability means a small window size where the probability to drop an event that matches the pattern is high and the probability to find an event as a replacement for the dropped event to match the pattern is low. On the other hand, with a higher match probability (i.e., a larger window size), the probability to drop an event that matches the pattern is low and the probability to find an event as a replacement for the dropped event to match the pattern is high. Hence, the percentage of false negatives decreases with a higher match probability. In the figure, the percentage of false negatives, for E-BL is 65% and 16% when the match probability is 6% and 89%, respectively. pSPICE reduces the percentage of false negatives by up to 300% compared to E-BL when the match probability is not too high. For a high match probability (cf. Figure 3.4a, in case match probability is 89%), E-BL outperforms pSPICE. However, please note, in CEP, it is unrealistic to have such a high match probability that implies completion of most PMs.

Figure 3.4b, using Q_2, shows similar behavior to the results of Q_1. The percentage of false negatives for pSPICE and PM-BL increases again with increasing match probability. However, pSPICE results in a lower percentage of false negatives by up to 58% compared to PM-BL till 81% match probability. After that, PM-BL outperforms pSPICE. This is because, as we mentioned above, all PMs have a high probability to complete and become complex events and hence it is hard for pSPICE to decide which PM to drop. Besides that, pSPICE has a slightly higher overhead than PM-BL which results in dropping more PMs and hence resulting in more false negatives. The results for E-BL are similar to the results in Q_1.

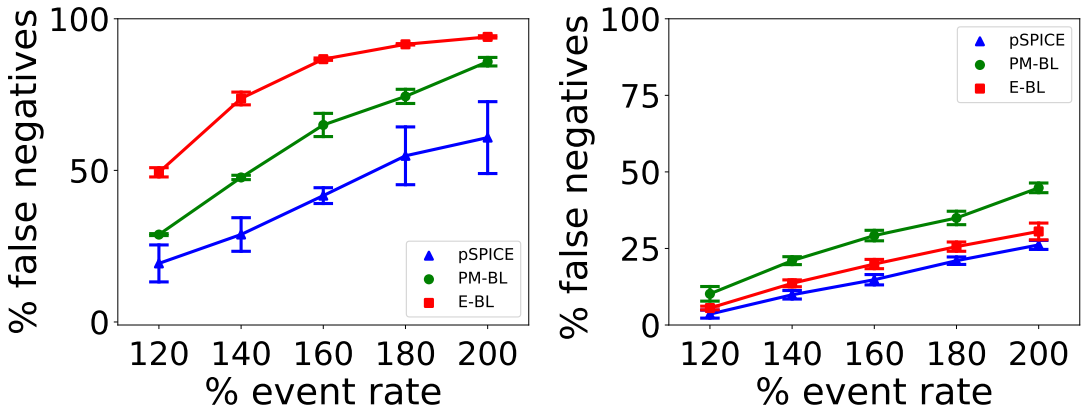
In Figure 3.4c, using Q_3, the percentage of false negatives produced by pSPICE and PM-BL also increases with increasing the match probability. pSPICE results in reducing the percentage of false negatives by up to 92% compared to PM-BL. As in Q_1 and Q_2, E-BL produces fewer false negatives when the match probability increases. A higher match probability in Q_3 means a smaller pattern size (in the figure, the match probability

50% corresponds to a pattern of size $n = 2$) which makes it easy to find a replacement event to match the pattern instead of a dropped event. The results for Q_3, compared to the results for Q_1 and Q_2, show that E-BL outperforms pSPICE with a smaller match probability (after 27%). This is because Q_3 uses *any* operator which means any event can match the pattern. Hence, the probability to find a replacement for a dropped event is much higher in Q_3 compared to Q_1 and Q_2 which matches a *sequence* of certain event types (stock symbol/company). Please note that, in Q_1 and Q_2, only the same event type can replace a dropped event of that type. Figure 3.4d, using Q_4, shows similar results to the results of Q_3 since the query of bus data is similar to the query of soccer data (i.e., Q_3). As a result, we skip explaining it.

Impact of event rate. To evaluate the impact of input event rate on the performance of pSPICE, we run experiments with Q_1, Q_2, Q_3, and Q_4 using the same setting as in the above section (cf. Section 3.3.2.1). However, to show the impact of different event rates, we streamed all datasets to the operator with event input rates that are higher than the maximum operator throughput by 20%, 40%, 60%, 80%, and 100% (i.e., event rate= 120%, 140%, 160%, 180%, 200% of the maximum operator throughput). In addition, we used a fixed match probability for all queries. Figure 3.5 depicts the impact of input event rates for Q_1 and Q_3, where the x-axis represents the event rate and the y-axis represents the percentage of false negatives. We use a match probability of 30% for Q_1 and 4% for Q_3. The results for Q_2 and Q_4 show similar behavior, hence we don't show them.

It is clear that using a higher event rate results in dropping more partial matches and hence increasing the percentage of false negatives. In Figure 3.5a, using Q_1, the percentage of false negatives for pSPICE increases with increasing the event rate, where it is 18.5% and 60% when the even rate is 120% and 200%, respectively. The same behavior is observed for PM-BL and E-BL. The percentage of false negatives for PM-BL increases from 29% to 86% and for E-BL from 49% to 94%, with the two event rates. Please note that for the considered match probability pSPICE is consistently better than PM-BL and E-BL, irrespective of the event rate. Figure 3.5b, using Q_3, as expected, shows similar behavior.

Maintaining LB. pSPICE performs load shedding to maintain a given latency bound. Figure 3.6 shows the result for running Q_2 with two event rates 120% (defined as $R1$) and 140% (defined as $R2$). In the figure, the x-axis represents time, and the y-axis represents the event latency l_e. We observed similar results for other event rates and queries and hence we don't show them. The figure shows that pSPICE always maintains the given latency bound LB which is 1 second in this experiment, regardless of the event rate.



(a) Q_1 **(b)** Q_3

Figure 3.5: Impact of event rate.

3.3.2.2 Impact of processing time (τ_γ) of a PM on utility calculation

Impact of processing time. As mentioned above, the completion probability P_γ of a partial match γ is a good indicator to know whether γ will complete or not. Therefore, we use it in calculating the utility of PMs (cf. Equation 3.1). However, the processing time τ_γ of a PM is also an important factor in calculating the utility of a PM. Therefore, we use it in deriving the utility of PMs as well (cf. Equation 3.1). To support this argument, we run experiments using pSPICE in two different ways of calculating the utility of PMs as follows: 1) using Equation 3.1, where we consider both the completion probability and processing time of PMs in calculating the utility of PMs and 2) considering only the completion probability in calculating the utility of PMs (i.e., the denominator in Equation 3.1 is 1). We refer to the load shedding strategy that considers only the completion probability in calculating the utility of PMs as pSPICE--.

To evaluate the performance of pSPICE and pSPICE--, we run both Q_1 and Q_2 in the same operator and use a window of size 10K and a pattern weight of one for both queries. The used event rate is 120%. Since we intend to analyze the impact of processing time in calculating the utility of PMs on QoR, we force the processing time of Q_1 to be higher than the processing time of Q_2 by a factor. We refer to this factor as τ_{Q_1}/τ_{Q_2}, where we use the following values: $\tau_{Q_1}/\tau_{Q_2} = 1, 2, 4, 8, 12, 16$. Figure 3.7 depicts the percentage of false negatives for pSPICE and pSPICE--. In the figure, the x-axis represents the factor τ_{Q_1}/τ_{Q_2} while the y-axis represents the percentage of false negatives.

In the figure, the performance of pSPICE and pSPICE-- is the same for low factors τ_{Q_1}/τ_{Q_2}. That is because the processing time of PMs in Q_1 and Q_2 has less impact on the utility. The difference between the percentage of false negatives between pSPICE and pSPICE-- increases when the factor τ_{Q_1}/τ_{Q_2} increases. The percentage of false negatives for pSPICE is 23% when $\tau_{Q_1}/\tau_{Q_2} = 16$ while it is 37.5% for pSPICE-- with the same factor. That shows that pSPICE results in reducing the percentage of false

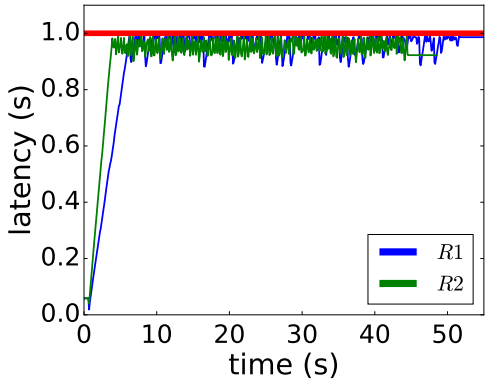


Figure 3.6: event latency l_e.

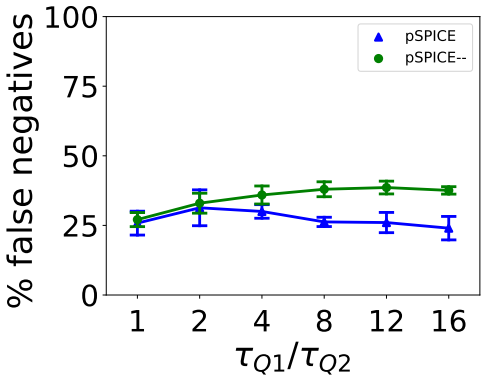


Figure 3.7: processing time τ_γ.

negatives by 62% compared to pSPICE-- for $\tau_{Q_1}/\tau_{Q_2} = 16$. As a result, we support our claim that considering the processing time of PMs is an important factor in calculating the utility of PMs.

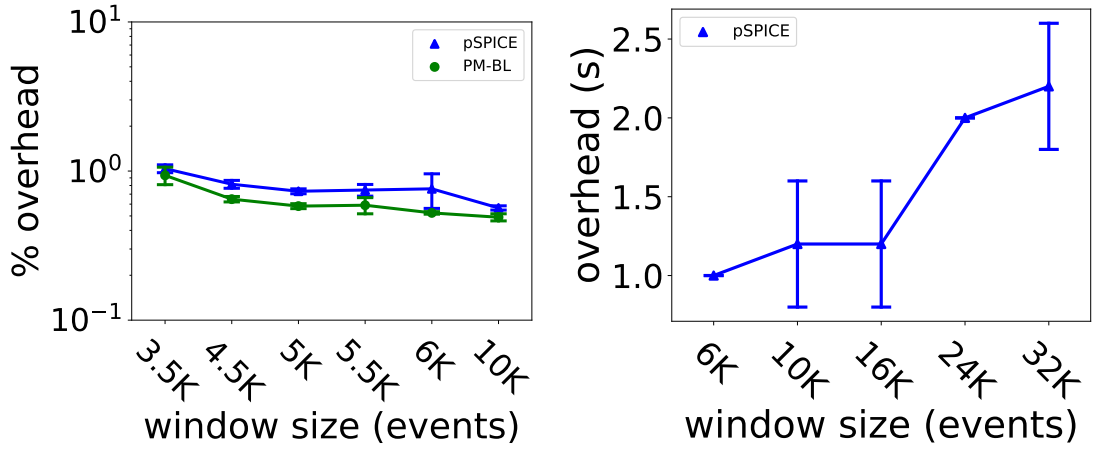
3.3.2.3 pSPICE overhead

Next, we show the overhead of pSPICE both during load shedding and during model building.

Load shedding overhead. The load shedder and the overload detector are time-critical tasks and their overhead directly affects QoR, therefore, they must be lightweight. To show the overhead of the load shedder and overload detector components in pSPICE, we run experiments with all queries using the same setting as in Section 3.3.2.1. Figure 3.8a depicts the results for Q_1, where the x-axis represents the used window size and the y-axis (log scale) represents the percentage of overhead compared to the total time that the operator needs to process the input dataset. We observed similar results for Q_2, Q_3, and Q_4 and hence we don't show them.

In the figure, the overhead of pSPICE is 1% in case the window size ws is 3.5K. The overhead of pSPICE decreases with increasing the window size, where the overhead is 0.7% when the window size is 10K. This is because a higher window size means that more windows are overlapped. Since events are processed in each window, the higher is the window overlap, the higher is the processing latency of events, and hence lower is the operator throughput. A low operator throughput results in having a smaller load shedding overhead as a percentage value. The overhead of PM-BL is slightly lower than the overhead of pSPICE which is expected since PM-BL performs random PMs shedding and doesn't have any cost for sorting PMs. This shows that pSPICE is a lightweight load shedding approach where its overhead is very low.

Model overhead. As we mentioned above, building the model is not a time-critical task. However, since there might be a need to retrain the model in case the distribution



(a) load shedding overhead **(b)** model overhead

Figure 3.8: overhead of pSPICE.

of input event stream and/or the content of input events change (cf. Section 3.2.4), we also analyze the overhead of building the model in pSPICE. An important factor that controls the overhead of building the model is the window size since it represents the number of iteration in the value iteration algorithm. Higher is the window size, more iterations are needed to solve the Markov reward process and hence higher is the overhead.

To evaluate the overhead of building the model, we run experiments with Q_1 with the same setting as in Section 3.3.2.1 but we use higher window sizes to show its impact on the overhead. We use the following window sizes: ws = 6K, 10K, 16K, 24K, 32K events. Figure 3.8b shows the overhead of model building in pSPICE, where the x-axis represents the window size and the y-axis represents the time needed in seconds. In the figure, as expected, the model building overhead increases with increasing the window size, where it is 1 second when the window size is 6K events and 2.4 seconds when the window size is 32K events. However, this overhead is still small which means that the model can be retrained without introducing a high overhead on the system or waiting a long time for a new model.

Discussion. Through extensive evaluations with several datasets and a set of representative queries, pSPICE shows that it has a very good performance w.r.t. QoR, where it usually outperforms both PM-BL and E-BL, especially with *sequence* operator and *sequence with repetition* operator. Only in the scenario of a relatively high match probability, E-BL might outperform pSPICE. However, since E-BL drops events, it might result in false positives, e.g., if the pattern contains the negation event operator (cf. Chapter 2, Section 2.2). Moreover, we show that pSPICE is a lightweight load shedding approach where its overhead is very low. The overhead of pSPICE is only slightly higher than the overhead of PM-BL.

3.4 Conclusion

In this chapter, we proposed an efficient, lightweight load shedding strategy, called pSPICE. In case of overload, pSPICE drops PMs from a CEP operator's internal state to maintain a given latency bound. To minimize the impact of load shedding on QoR, we proposed to utilize two important features (i.e., current state of a PM and number of remaining events in a window) that reflect the importance of PMs and used these features in calculating the utility of PMs, where we model the pattern matching operation as a Markov reward process. By thoroughly evaluating pSPICE with three real-world datasets and multiple important queries in CEP, we show that pSPICE considerably reduces the degradation in QoR compared to state-of-the-art load shedding strategies.

Chapter 4

eSPICE: Probabilistic Load Shedding from Input Event Streams

In the previous chapter (cf. Chapter 3), we presented pSPICE, a *white-box* load shedding approach that drops PMs, i.e., it sheds load on the PM granularity. As we showed, dropping PMs might be performed in an efficient and lightweight manner. Moreover, dropping PMs with low utilities reduces the adverse impact of shedding on QoR while saving the processing power and enabling a CEP operator to maintain a given latency bound. However, if the match probability of PMs is high, pSPICE is forced to drop important PMs that might complete and become complex events, hence adversely impacting QoR (cf. Section 3.3.2). Therefore, in this chapter, we present a *black-box* load shedding approach that drops events from windows in the input queue of a CEP operator, i.e., it sheds events on the window granularity. As we mentioned in Section 2.1, the input event stream of a CEP operator is partitioned into windows. In a window, there might exist many events that have no or only little influence on the detected complex events within the window. Hence, dropping these events might have a low negative impact on QoR.

Dropping events from windows in the input queue of a CEP operator reduces the load on the operator, hence enabling the operator to maintain a given latency bound. However, dropping events might also adversely impact QoR where it might result in false positives and negatives. Thus, it is crucial to drop those events that have less impact on QoR, i.e., drop events that have low utilities. In CEP systems, there are primarily three challenges facing the decision to drop events from windows: 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 as an event may be dropped from some windows while it is still there in other windows since windows may overlap. 3) Dropping events in an efficient way to reduce the overhead of load shedding.

In this chapter, we propose a *black-box* load shedding approach, called eSPICE, for CEP systems. eSPICE is an efficient and lightweight approach that sheds *events from windows*, i.e., it sheds events on the window granularity. Moreover, it considers the dependency between events of the same pattern as well as the order of events in the pattern and in the input event stream. In addition, it also considers the impact of the same event residing in overlapping windows on QoR, where the same event may be in different positions within 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 main contributions in this chapter are as follows:

- We propose an efficient lightweight load shedding strategy, called eSPICE, that uses a probabilistic model to capture the utility of events in a window. The utility of an event is influenced by its type and its relative position within 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 be performed.

- 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. Additionally, we compare the performance, w.r.t. QoR, of eSPICE with state-of-the-art load shedding strategies.

The rest of the chapter is structured as follows. Section 4.1 presents the used system model. In Section 4.2, we provide a detailed explanation of how the different components of eSPICE are used for our probabilistic load shedding strategy. Section 4.3 presents results obtained from extensively evaluating eSPICE. Finally, we conclude this chapter in Section 4.4.

4.1 System Model

We use the same system model as presented in Chapter 2, Section 2.1, where we assume a window-based CEP system that consists of a single operator. The operator matches a set of patterns $\mathbb{Q}$ where each pattern $q_i \in \mathbb{Q}$ has a corresponding weight $w_{q_i} \in \mathbb{W}_{\mathbb{Q}}$, reflecting its importance. Moreover, we assume a *black-box* operator, where we do not have knowledge about the operator's internal state (i.e., PMs) and the used computational

model. 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 where set $\mathbb{T} = \{T_1, T_2, ..., T_m\}$ represents the set of all event types in the input event stream.

4.2 Probabilistic Load Shedding

To minimize the degradation in the quality of results, our main idea is to avoid dropping events that could contribute to producing complex events. The question is– how do we identify the utility of these events before processing them? In real-world applications, event streams have properties that can be exploited to derive the aforementioned utility of an 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 events that contribute to complex events. For example, in a soccer game, a sports 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, the position of the events contributing to the complex events, i.e., the 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 that detects such an influence. Again, in a different domain, the sensor data set provided by the Intel Research Berkeley Lab shows a positive correlation between events of type temperature and events of type humidity [Bho+18]. This implies that within a certain interval an increase/decrease in temperature results in an increase/decrease in humidity.

Moreover, an important event operator in CEP is the sequence operator [Sad+04; RLR16; LG15]. In the sequence operator, different event types in a pattern have different importance in different window positions. For example, let us assume a pattern $q = seq(A; B; C)$. At the beginning of a window, an event instance of type A has a higher match probability than an event instance of type C, i.e., a higher probability to contribute to PMs, hence to contribute to complex events. Therefore, in pattern q, events of type A have higher importance than events of type C at the beginning of windows. That shows that in the sequence operator, the probability of an event to be part of complex events might depend on its type and relative position within windows. As a result, we exploit this correlation, captured by the type and relative position in the window of events, to predict the probability of events to contribute to a complex

event(s). In particular, we derive utilities of 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 within 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 events within windows. That 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.

4.2.1 The eSPICE Architecture

To enable load shedding, similar to Chapter 3 (i.e., pSPICE), we extend the architecture of a CEP operator by adding the following components– overload detector, model, and load shedder (LS)– as depicted in Figure 4.1. The functionalities of these components are similar to their functionalities, as explained in pSPICE (cf. Chapter 3). However, the way these components work in eSPICE is different from how they work in pSPICE, as in eSPICE, we drop events, not PMs. 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 events are queued. In case of an overload, the LS drops events from windows to prevent the violation of the defined latency bound (LB). The model contains the utility of events in a window and other information that is needed by the LS. Later in this chapter, we explain, in detail, how these three components work in eSPICE.

Now, we explain how these three 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 events in a window, available from the model, 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 in this chapter, our load shedder has very low overhead. Contrarily, building the model can afford to be computationally heavy as it is not a time-critical task.

4.2.2 Utility Model and Its Application

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

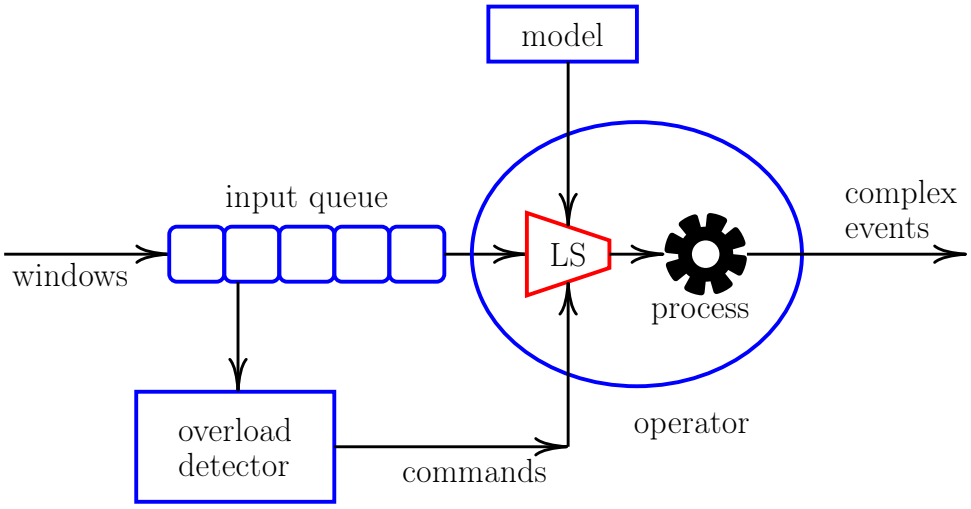


Figure 4.1: The eSPICE Architecture.

4.2.2.1 Utility Prediction Function

The utility of an event in a window is defined by its impact on QoR. As mentioned earlier, we represent utility as the probability of the event to contribute to a complex event(s). Clearly, dropping events that have a high probability to contribute to complex events degrades QoR. Hence, we avoid dropping these events by assigning high utility values to them. Please recall that we identified the **type** $T_e \in \mathbb{T}$ and **position** P_e of event e within a window to be an indicator of whether or not this event has a high probability to contribute to complex events. This implies that the type and position of an event determine its utility. As a result, to map the type and position of events to a utility value, we introduce the utility prediction function in Equation 4.1:

$$U_e = f(T_e, P_e) \tag{4.1}$$

that predicts the utility U_e of event e of type $T_e \in \mathbb{T}$ at position P_e within a window. As we will see later, this prediction function can be simply implemented based on statistical data collected from the operator.

4.2.2.2 Utility Threshold and Occurrences

Upon receiving the drop command to drop ρ events from each window, the LS must find those ρ 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 $O(ws.log_2(\rho))$, where ws is the number of events in a window [Ski08]. However, this approach requires that the entire window is available to the LS before sorting of the utilities and consequently shedding of events is 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 overload.

A reasonable 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 (ρ) to a utility threshold u_{th}, i.e., $f : \rho \rightarrow u_{th}$. To find the utility threshold u_{th}, we could predict the number of ρ event occurrences in a window, whose utility is less or equal to the utility threshold u_{th}.

More specifically, in window w, we define the number of event occurrences, whose utility is less or equal to a certain utility value u as follows: $O_u = |\{e : U_e \leq u\}|$. The number of event occurrences O_u in 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 shorthand, 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.

4.2.2.3 Applying Utility Models in Load Shedding

Now, we describe how load shedding is performed in eSPICE. To drop ρ events from each incoming window, the LS first searches for the cumulative utility occurrences O_u, which has a value $O_u \geq \rho$. Then, the LS uses the utility value u as a utility threshold u_{th} to drop those ρ 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_e of the event e in w using the utility prediction function $f(T_e, P_e)$ (cf. Equation 4.1). If the event utility U_e in window w is greater than the utility threshold u_{th}, the LS keeps event e in window w. Otherwise, it drops event e from widow w. The utility threshold u_{th} enables the LS to drop ρ events from each window.

4.2.3 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, i.e., $\mathbb{T} = \{A, B\}$ (cf. Figure 4.3).

4.2.3.1 Building the Utility Prediction Function

As mentioned above (cf. Section 4.2.2), the utility U_e of event e is represented by the probability of the event to contribute to the detected complex events. To predict the utility of events in a window, we collect statistics, from the already detected complex events, on the types T_e and relative positions P_e within windows of *events* that contributed to those detected complex events. More specifically, we count the number of occurrences of each event type $T_e \in \mathbb{T}$ at each position P_e in a window that contributed to the detected complex events. The number of occurrences of events within detected complex events provides an insight into the importance (or utility) of the event types and their relative positions within a window. The operator might match multiple patterns (i.e., multi-patterns operator), where each pattern $q_i \in \mathbb{Q}$ has its corresponding weight w_{q_i} (cf. Section 4.1). Therefore, when counting the number of occurrences of events from detected complex events, we multiply the number of occurrences of events by the corresponding pattern weight to which the detected complex event belongs.

As a result, we simply normalize those number of occurrences to generate the utility U_e (i.e., implement the utility prediction function $f(T_e, P_e)$ in Equation 4.1) of event e of a certain event type T_e at a certain window position P_e. 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 (i.e., $M = |\mathbb{T}|$) and N represents the window size ws. Each of its cells $UT(T_e, P)$ represents the utility of a specific *event type* $T_e \in \mathbb{T}$ in a certain *position* P_e in a window, where the utility value U_e of event e is stored in $UT(T_e, P)$. The values in UT could be too fine-grained. We limit the number of different utility values by normalizing the values in UT between 0 and 100 and rounding them to integers, i.e., $UT(T_e, P) \in [0, 100]$. Referring to our above example, Table 4.1 shows a utility table that is generated from the collected statistical data.

4.2.3.2 Building Utility Threshold and Occurrences

As we discussed in Section 4.2.2.2, to drop ρ events from each window, we should find a utility threshold u_{th} that results in dropping ρ 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 ρ events in a window, i.e., $O_u \geq \rho$. Then, we use u as a utility threshold u_{th} to drop ρ events from each window.

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.

T_e/P_e	0	1	2	3	4
A	70	15	10	5	0
B	0	60	30	10	0

Table 4.1: UT generated from the collected statistical data.

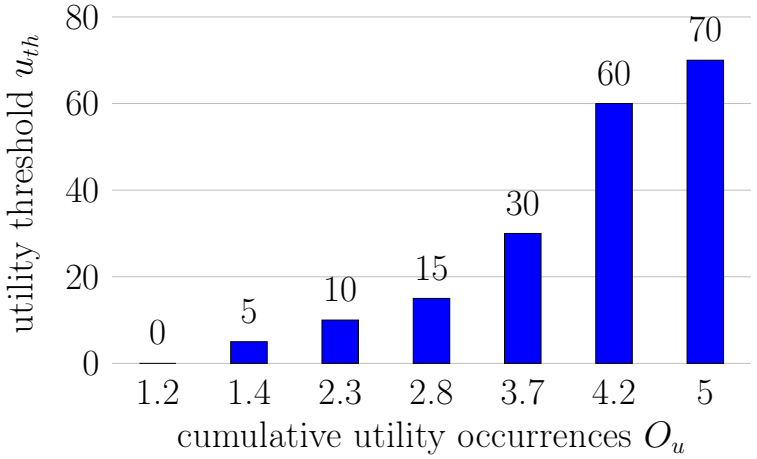


Figure 4.2: CDT computed from Table 4.1 (UT) and the predicted position shares in a widnow.

Figure 4.3: Simple running example.

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 $UT(T_e, P) \in [0, 100]$), 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:

$$O_u = \begin{cases} o_u, & \text{if } u = 0 \\ o_u + O_{(u-1)}, & \text{otherwise} \end{cases} \tag{4.2}$$

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 *every single position* in UT, there exists a utility value for each event type. For example, in Table 4.1, every 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, $UT(A, 0) = 70$ and $UT(B, 0) = 0$. 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 events within the window. The position shares in a window $S(T_e, P)$ of event e of type T_e at position P_e in the window equals the probability of this event type T_e to come at position P_e 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 $U_e = UT(T_e, P)$ for the event type T_e at position P_e in UT, we increase the number of occurrences o_{u_e} by $S(T_e, P)$. The cumulative utility occurrences O_u is then computed as in the case of a single event type using equation 4.2. 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} that drops ρ events from each window, we iterate over CDT to find a cell value $CDT(u)$ that is $\geq \rho$, which means that the number of events with utility values less or equal to u occurs at least ρ times in each window. Hence, using u as a utility threshold drops at least ρ events from each window. We explain the utility threshold prediction further with the help of our running example. Figure 4.2 shows the CDT computed from UT in Table 4.1 and the predicted position shares in a window. Now, to drop $\rho = 2$ events from each window, in the figure, $CDT(10) = 2.3 > \rho$. Thus, to drop $\rho = 2$ events from each window, we use the utility threshold $u_{th} = 10$.

Algorithm 3 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 $u = UT(T_e, P)$, i.e., the utility of the event type T_e 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 $S(T_e, P)$. Since the utility values are used as indices in CDT, they are already sorted in 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 (6-8)).

Algorithm 3 Building CDT table.

```
1: computeCDT () begin
2: for T_e ∈ 𝕋 do
3:     for P = 0  to  (N − 1) do          // N: the window size ws.
4:         u = UT(T_e, P)
5:         temp(u) += S(T_e, P)           // temp(u) = o_u

   // accumulate utility values in ascending order.
6: CDT(0) = temp(0)
7: for u = 1  to  100 do
8:     CDT(u) = temp(u) + CDT(u − 1)
9: end function
```

4.2.4 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 (l_e). It compares l_e with the defined latency bound LB and decides to drop events if LB might be violated. Recall that 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. The operator throughput μ represents the maximum number of events the operator can process per second, i.e., the maximum service rate. 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 event e at 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 4.4. 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.

4.2.4.1 Dropping Interval

So far, we have considered dropping ρ 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 4.4). More specifically, we need to drop ρ 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 ρ 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 ρ 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 4.4) and drop ρ 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, the larger the partition size is, the 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 that is as large as possible (of course, the upper bound being the buffer size). More specifically, we partition a window in β partitions of equal sizes, where $\beta = ceil(\frac{ws}{q_{max} - f.q_{max}})$. As a result, the partition size $p_{size} = \frac{ws}{\beta}$. We use the partition size as a *dropping interval* in which ρ 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 ρ 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 $\beta > 1$), 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 ρ events from each partition of the incoming windows, each partition has its own utility threshold u_{th}.

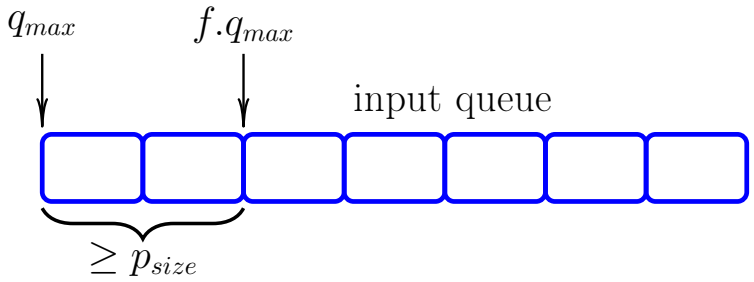


Figure 4.4: Partition Size.

4.2.4.2 Dropping Amount

The dropping amount represents the number of events that must be dropped from each partition of each window. Determining how many events ρ to drop per partition depends on the input event rate R and the operator throughput μ. The overload detector first computes the overload ratio (denoted by δ) as follows: $\delta = 1 - \frac{\mu}{R}$. Then, the number of events ρ to drop per partition is computed as follows: $\rho = \delta.p_{size}$.

4.2.4.3 Appropriate f Value

As we mentioned above, using a high f value prevents dropping events in short burst cases, hence decreases the degradation in the quality of results. However, the f value controls the partition size p_{size}, 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 having a partition size that 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 exist at least ρ 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.

4.2.5 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_e may be different in different windows since the event position P_e 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 ($\beta = 1$), 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_e from UT and also determines the partition ($part$) in a window to which event e belongs, both in $O(1)$ time-complexity. Then, the LS compares the event utility U_e with the utility threshold u_{th} of the corresponding partition ($part$) to decide on whether or not to drop event e from the window. If the utility U_e of event e is less or equal to the utility threshold of its corresponding partition, the LS drops event e.

Algorithm 4 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 ρ to drop per partition to LS. The LS receives drop commands from the overload detector where it first calculates the utility threshold u_{th} for each partition depending on the required number of events ρ 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 $\geq \rho$ (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 U_e 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.

4.2.6 Extensions

In this section, we explain the following extensions to our load shedding approach: handling a variable window size, using bins for large windows, retraining the model, and supporting the negation operator [WDR06]. Handling a variable window size enables our approach to work with windows of different sizes. While the use of bins enables eSPICE to work with large windows.

Algorithm 4 Load shedder.

```
1: getUtilityThresholdForEachPartition (ρ) begin
2:  for part = 0  to  (β − 1) do
3:      for u = 0  to  100 do
4:          if CDT(part, u) ≥ ρ then
5:              u_th(part) = u
6:              break // break the inner loop and proceed to the next partition.

7: end function

8:  applyLS (e) begin      // Event e of type T_e at position P_e in the window
9:  if !LS.isActive then
10:     return  false
11:  else
12:     part = P_e / p_size
13:     if UT(T_e, P_e) ≤ u_th(part) then
14:         return  true
15:     else
16:         return  false

17: end function
```

4.2.6.1 Handling Variable Window Size

The incoming windows might have a variable window size ws depending on the window splitting strategies. As mentioned in Section 2.1, 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 [MTR17].

As explained earlier, in order to implement the utility prediction function, we use table 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 observed window size. Since N might be different from the size of actual windows, 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 w, if $ws > N$,

we scale down window w where more than one position in window w is mapped to a single position in UT. On the other hand, If $ws < N$, we scale up window w where each position in window 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 window w are 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 the 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 window 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 the 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, in the case of time-based and pattern-based windows (cf. Section 2.1), the actual window size is unknown at the time when the LS performs a lookup in UT to get the utility of an event in a window based on its relative position. However, it is impossible to get a relative event position if the actual window size is unknown. Yet, the window size is important for the lookup, and we must predict it. For example, in the case of a time-based sliding window, the input event rate could be used to predict the window size. If the distributions of events within windows are known, we may precisely predict the window size. However, if the event distributions are unknown or continuously changing, it becomes hard to predict the actual window size, which might negatively impact the performance, w.r.t. QoR, of eSPICE. Please note that predicting the window size is already researched in literature [MTR17] and will not be the focus of this work.

4.2.6.2 Using Bins for a Large Window Size

The average observed 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 4.3, we discuss more the impact of the bin size on the quality of results.

4.2.6.3 Model Retraining

The distribution of events in the input event stream may change over time which may influence the accuracy of the constructed model (i.e., the constructed utility table UT and CDT table). That may adversely impact QoR. Therefore, in this case, we must retrain the model to capture these changes in the input event stream. We might either retrain the model periodically or only when the input event stream changes. Two factors may indicate that the input event stream has changed: 1) the change in the distribution of events in the input event stream (i.e., the distribution of event types), and 2) the change in the distribution of the event content (i.e., the event's actual data) in the input event stream. As a result, to maintain the constructed model accurately, eSPICE must retrain the model if *at least* one of these two factors has changed, i.e., (the event distribution or/and the event content distribution). eSPICE periodically gathers statistics from windows, hence if there is a need to retrain the model, eSPICE uses these gathered statistics to build a new model.

These two factors are application specific where one or both of these factors may change over time depending on the application. For example, in applications, where input event streams are generated in a fixed frequency, the distribution of input events is fixed, however, the distribution of event contents may change over time. An example is a transportation application, where bus events from different buses are generated at a fixed frequency, however, the content of bus events may change over time depending on the time of day. On the other hand, for some other applications, the event distribution may change while the event content distribution may stay fixed or only slightly change. For example, an ID reader in a retail shop generates an ID event whenever an item is scanned. During peak hours, more items may be scanned which changes the distribution of ID events, however, the distribution of the content of ID events might only slightly change– only when the item's content changes (e.g., the item price changes). Finally, both event distribution and event content distribution may change over time. For example, in a stock application, a stock event of company A may be generated only if the stock quote of company A has changed which implies that the stock event distribution might change over time depending on the change in the stock quotes. Additionally, the change in stock quotes might depend on multiple factors, where a stock quote changes if one or more of these factors change, i.e., the distribution of stock event content may change over time.

The number of detected complex events within windows gives a good insight into the event distribution and event content distribution. If the average number of detected complex events within windows changes, this provides a good indication that the event distribution or the event content distribution has changed. Therefore, to capture the distribution changes, we use the following approach. We compute the average number of complex events per window during model building. Then, we periodically compare this average number with the average number of complex events within the newly coming windows that are marked for statistic gathering, i.e., those windows from which the LS

does not drop events. If the deviation between both averages is higher than a threshold, eSPICE should rebuild the model. This approach is lightweight and efficient, where its overhead is very low.

4.2.6.4 Supporting Negation Operator

As discussed above (cf. Section 4.2.2), eSPICE learns about the utility of events within a window depending on the detected complex events within already processed windows. However, in the case of the negation operator, a complex event is detected only if specific event type(s) are not present in a window. For example, let us assume that an operator matches pattern $q = seq(A; !B; C)$. In pattern q, we call event type B a *negated event type*. In a window, while matching pattern q, the operator detects a complex event if an instance of event type A happens, followed by an instance of event type C. However, in case, an instance of event type B happens after the occurrence of an instance of event type A and before the occurrence of event type C, no complex event is detected, and the already matched part of pattern q is ignored– called *abandoned partial match*. Hence, eSPICE cannot learn about the importance of the negated event type(s) (i.e., B in pattern q) since the operator does not produce any complex event with event type B. That means, in this example, the instances of negated event type B in windows will be assigned a utility value zero. Hence, if there is overload and the LS must drop events, it will start to drop events of type B since event B has a utility value of zero. That might cause many false positives if events of event types A and C are present in windows, while events of type B are dropped.

To avoid the above problem and to enable eSPICE to learn about the importance of the negated event types in an operator, we request the operator to forward the abandoned PMs to eSPICE, where eSPICE gathers statistics from these abandoned PMs, as well. This way, eSPICE, as for any other event type, can gather statistics on the importance of the negated event types in different positions within windows using these abandoned PMs, i.e., learns about the utility of those negated event types. As a result, eSPICE may avoid dropping instances of the negated event types, hence reducing the number of false positives.

4.3 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 μ.

4.3.1 Experimental Setup

Here, we describe the evaluation platform, the baseline implementation, datasets, and queries used in the evaluations. In this chapter, we use the same evaluation platform as in Chapter 3, Section 3.3.1. We compare the performance, w.r.t. QoR, of eSPICE

with the performance of pSPICE and E-BL (cf. Chapter 3, Section 3.3.1) where we rename E-BL to BL in this chapter. As evaluation results showed that a completely random event shedder is comprehensively outperformed by eSPICE, we do not show the evaluation results for a completely random shedder.

Datasets. We use two of the datasets explained in Section 3.3.1. In particular, we use the stock quotes stream from the New York Stock Exchange (i.e., the *NYSE Stock Quotes* dataset) and the position data stream from a real-time locating system in a soccer game (i.e., the *RTLS* dataset). We do not show evaluation results on the public bus transport (i.e., the *PLBT* dataset) from the previous chapter since the evaluation results produced by performing event shedding on the *PLBT* dataset are similar to those results when performing event shedding on the *RTLS* dataset (cf. Section 3.3.2).

Queries. We employ six queries (Q_1, Q_2, Q_3, Q_4, Q_5, and Q_6) that cover an important set of operators in CEP as shown in Table 4.2: sequence operator, sequence operator with repetition (which also contains Kleene closure), disjunction operator, sequence with negation operator, and sequence with any operator [ZDI14; CM94; CM10; WDR06; MM09]. In Table 4.2, C_i represents the stock quota of company i, and D_i represents the event of player i. We use the same window strategies (i.e., time-based and count-based windows) and the same selection and consumption policies used in Section 3.3.1. The queries Q_1, Q_2, and Q_6 are the same as queries Q_1, Q_2, and Q_3, respectively, defined in Section 3.3.1. Moreover, we add three more queries, namely Q_3, Q_4, and Q_5, to cover a broader range of CEP queries. Next, we explain all these six queries where we re-explain queries Q_1, Q_2, and Q_6, here again, to make the presentation smoother.

Q_1 (sequence operator) detects a complex event when rising or falling stock quotes, by a given percentage, of 10 certain stock symbols are detected within *ws* events/minutes in a certain sequence. Q_2 (sequence operator with repetition) detects a complex event when 10 rising or 10 falling stock quotes, by a given percentage, of certain stock symbols *with repetition* are detected within *ws* events/minutes in a certain sequence. Q_3 (multi-pattern operator) detects a complex event if either Q_1 or Q_2 matches. Q_3 represents a multi-pattern operator. Q_4 (sequence with negation operator) 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 (sequence with any operator) detects a complex event when any 20 rising or any 20 falling stock quotes, by a given percentage, of any stock symbol are detected within *ws* seconds/minutes from a rising or falling quote of a leading stock symbol (defined as MLE). The leading stock symbols are composed of a list of 4 technology blue-chip companies. Q_6 (sequence with any operator) uses the RTLS dataset. It detects a complex event when any n players of a team 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. We use two players as strikers, one striker from each team.

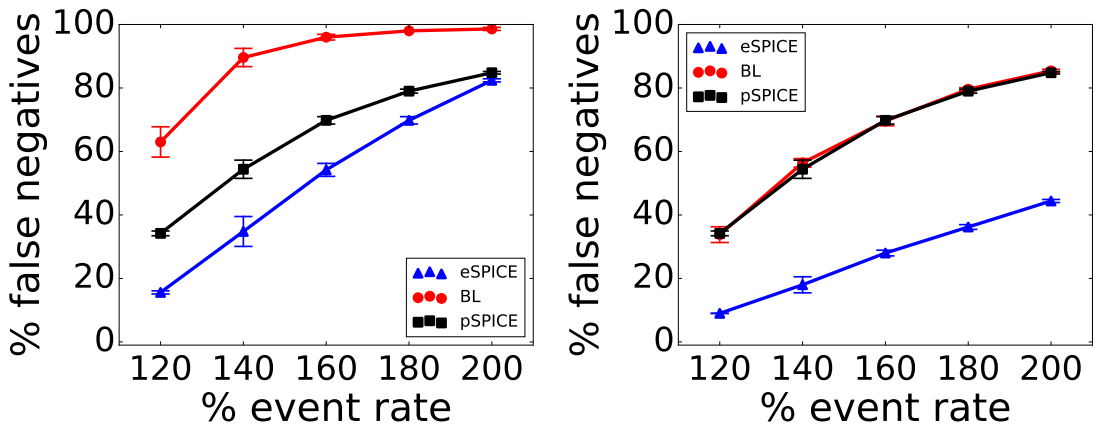
Stock queries	
Q_1	**pattern seq**$(C_1; C_2; ..; C_{10})$ **where** *all* C_i *rise by* $x\%$ **or** *all* C_i *fall by* $x\%$, $i = 1..10$ **within** ws minutes/events
Q_2	**pattern seq**$(C_1; C_1; C_2; C_3; C_2; C_4; C_2; C_5; C_6; C_7; C_2; C_8; C_9; C_{10})$ **where** *all* C_i *rise by* $x\%$ **or** *all* C_i *fall by* $x\%$, $i = 1..10$ **within** ws minutes/events
Q_3	$\mathbf{Q_1} \vee \mathbf{Q_2}$
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/events
Q_5	**pattern seq**$(MLE;$ **any**$(20, C_1; C_2; ..; C_{20}))$ **where** MLE *rises by* $x\%$ **and** *all* C_i *rise by* $x\%$ **or** MLE *falls by* $x\%$ **and** *all* C_i *fall by* $x\%$, $i = 1..20$ **within** ws seconds/minutes
Soccer query	
Q_6	**pattern seq**$(S;$ **any**$(n, D_1, D_2, .., D_m))$ **where** S *possesses ball* **and** $distance(S, D_i) \leq x$ *meters* , $i = 1..m$ and m is the number of players in a team **within** ws seconds

Table 4.2: Queries.

4.3.2 Experimental Results

In this section, we evaluate the impact of our probabilistic load shedding strategy (eSPICE) on QoR, particularly the number of false positives and false negatives, and compare its results with the results of BL and pSPICE. Moreover, we show the impact of the window size and the bin size on QoR. Additionally, we analyze the overhead of eSPICE and show its ability to maintain the given latency bound.

If not noted otherwise, we employ the following settings. Q_1, Q_2, Q_3, and Q_4 use a count-based sliding window. While we use a time-based sliding window for Q_5 and Q_6. For Q_1, Q_4, Q_5, and Q_6, a logical predicate is used to open new windows. In Q_1, Q_4, and Q_5, a new window is opened for each incoming event of the leading stock symbols (MLE), while, in Q_6, a new window is opened for each incoming striker event (S). For Q_2 and Q_3, a count-based predicate is used where a new window is opened every 20 events, i.e., the slide size equals 20 events. The number of defenders in query Q_6 is 4, i.e., $n = 4$. 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 operator throughput μ until the model is built. After that, we



(a) strict QoR **(b)** relaxed QoR

Figure 4.5: False negatives for Q_1 with different input event rates.

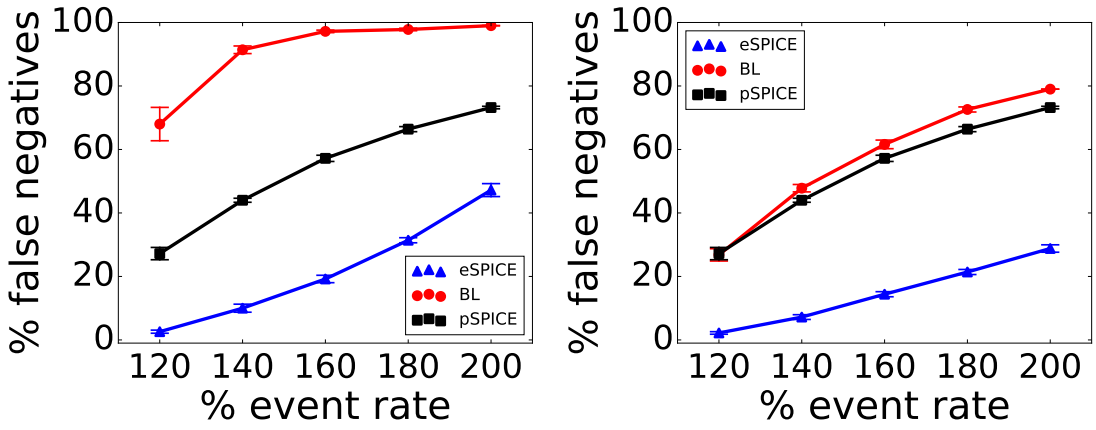
increase the input event rate to enforce load shedding as we will mention in the following experiments. We execute several runs for each experiment and show the mean value and standard deviation.

There are several factors that may influence the performance, w.r.t. QoR, of eSPICE such as the event rate, the window size, and the bin size. Next, we evaluate the performance of eSPICE with these different factors. Moreover, since we have two approaches to define QoR, namely the strict QoR and relaxed QoR, we also show the results when using both QoR approaches.

4.3.2.1 Impact of event rate on QoR

Next, we analyze the impact of eSPICE on QoR (i.e., the number of false negatives and positives) with different input event rates. To show the impact of event rate on QoR, we stream the datasets to the operator with input event rates that are higher than the operator throughput μ by 20%, 40%, 60%, 80%, 100% (i.e., event rate= 120%, 140%, 160%, 180%, 200% of the operator throughput μ). For Q_1, Q_2, and Q_3, we use a window of size 1200 events (i.e., $ws = 1200$). The used window sizes for Q_4, Q_5, and Q_6 are 600 events, 20 minutes, and 30 seconds, respectively. As we showed in Chapter 3, Section 3.3, the match probability has a considerable impact on the performance (w.r.t. QoR) of load shedders, where the match probability is computed from the ground-truth by dividing the total number of complex events by the total number of PMs. The match probability for Q_1, Q_2, Q_3, Q_4, Q_5, and Q_6 are 32%, 22%, 27%, 66%, 14%, and 6%, respectively.

Number of false negatives. Figures 4.5, 4.6, 4.7, 4.8, 4.9, and 4.10 depict the impact of event rates on QoR. The figures show the percentage of false negatives for all queries (Q_1, Q_2, Q_3, Q_4, Q_5, and Q_6). In the figures, the x-axis represents the input event rate



(a) strict QoR **(b)** relaxed QoR

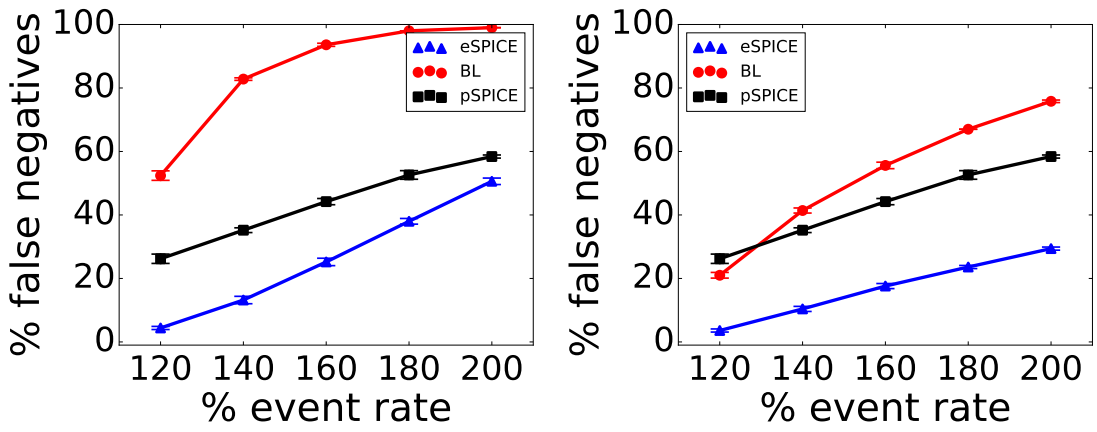
Figure 4.6: False negatives for Q_2 with different input event rates.

and the y-axis represents the percentage of false negatives.

The percentage of false negatives increases if the input event rate increases since more events/partial matches must be dropped. Figure 4.5a shows the percentage of false negatives for Q_1 using strict QoR. In the figure, the percentage of false negatives caused by eSPICE increases from 16% to 82% when increasing the event rate from 120% to 200%, respectively. Whereas, the percentage of false negatives caused by BL and pSPICE increases from 67% to 98% and from 35% to 84% when increasing the event rate from 120% to 200%, respectively. The results in Figure 4.5a show that eSPICE outperforms, w.r.t. QoR, both BL and pSPICE. The results for all load shedders in the case of relaxed QoR (as depicted in Figure 4.5b) show similar behavior to the results when using strict QoR. However, in the case of relaxed QoR, eSPICE and BL result in a lower percentage of false negatives compared to the case of strict QoR. The reason behind this is that in the strict QoR, only certain event instances are allowed to match the pattern and produce complex events, otherwise, the produced complex events are considered as false negatives/positives. As depicted in the figure, the impact of pSPICE using relaxed QoR is similar to its impact when using strict QoR. Here again, eSPICE outperforms the other load shedders when using relaxed QoR.

Figures 4.6a and 4.6b depict the percentage of false negatives for Q_2 when using strict and relaxed QoR, respectively. In Figure 4.6a, the percentage of false negatives caused by all load shedders increases when the input event rate increases. However, eSPICE performs better than the other load shedders irrespective of the input event rate where eSPICE outperforms BL and pSPICE by up to 21 and 10 times, respectively. The results in Figure 4.6b show a similar behavior where again eSPICE outperforms the other load shedders. We conclude from the results shown in Figures 4.5 and 4.6 that when using the sequence operator (i.e., Q_1 and Q_2), eSPICE always performs, w.r.t. QoR, better than BL and pSPICE regardless of the used input event rate and the way QoR is calculated.

The results depicted in Figure 4.7 show the performance of the load shedders when



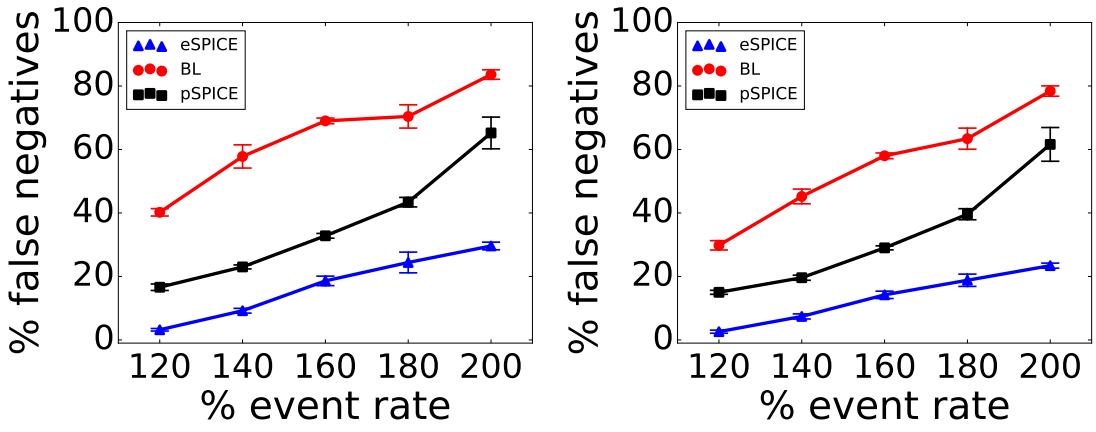
(a) strict QoR **(b)** relaxed QoR

Figure 4.7: False negatives for Q_3 with different input event rates.

using a multi-pattern operator (i.e., Q_3). Figures 4.7a and 4.7b show the results when using strict and relaxed QoR, respectively. The percentage of false negatives again increases when the input event rate increases. The percentage of false negatives caused by eSPICE, BL, and pSPICE increases from 5% to 51%, from 51% to 99%, and 25% to 58% when increasing the input event rate from 120% to 200%, respectively. This shows that eSPICE outperforms both BL and pSPICE up to 10 and 5 times, respectively. The same behavior is observed in Figure 4.7b where eSPICE performs better than the other load shedders. This shows that eSPICE supports the multi-pattern operator with a considerably low adverse impact on QoR.

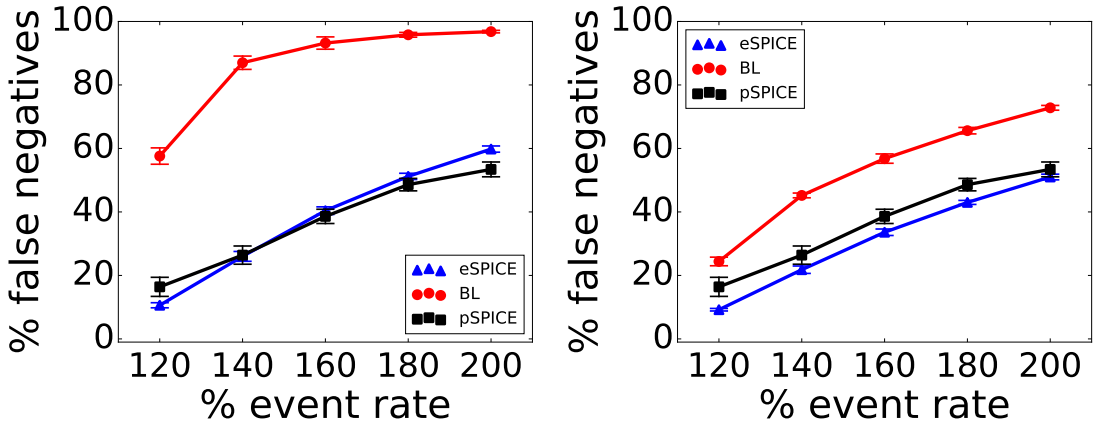
To evaluate the performance of eSPICE with the negation event operator, we run experiments with Q_4. In Q_4, we limit the number of complex events to only one event per window. The window is closed if a complex event is detected. We do that to determine the impact of the negation operator on the matching output. The results with both strict and relaxed QoR are depicted in Figures 4.8a and 4.8b, respectively. Again, with the negation operator, eSPICE outperforms both BL and pSPICE using strict or relaxed QoR. The performance of eSPICE is better than the performance of BL and pSPICE by up to 13 and 6 times when using strict QoR and by up to 10 and 6 times when using relaxed QoR. That shows that eSPICE supports the negation operator with a considerably low negative impact on QoR.

Finally, we evaluate the performance of eSPICE with the *any* event operator where the results are depicted in Figures 4.9 and 4.10 which show results for Q_5 and Q_6, respectively. Figure 4.9a shows results for Q_5 using strict QoR. In the figure, the percentage of false negatives caused by all load shedders increases when the input event rate increases. The figure shows that eSPICE performs better than BL. However, the performance of eSPICE is worse than the performance of pSPICE for the majority of input event rates. The reason behind this is that the even utilities in Q_5 are spread and less accurately predicted since Q_5 represents an *any* operator where Q_5 matches an event of any type



(a) strict QoR **(b)** relaxed QoR

Figure 4.8: False negatives for Q_4 with different input event rates.

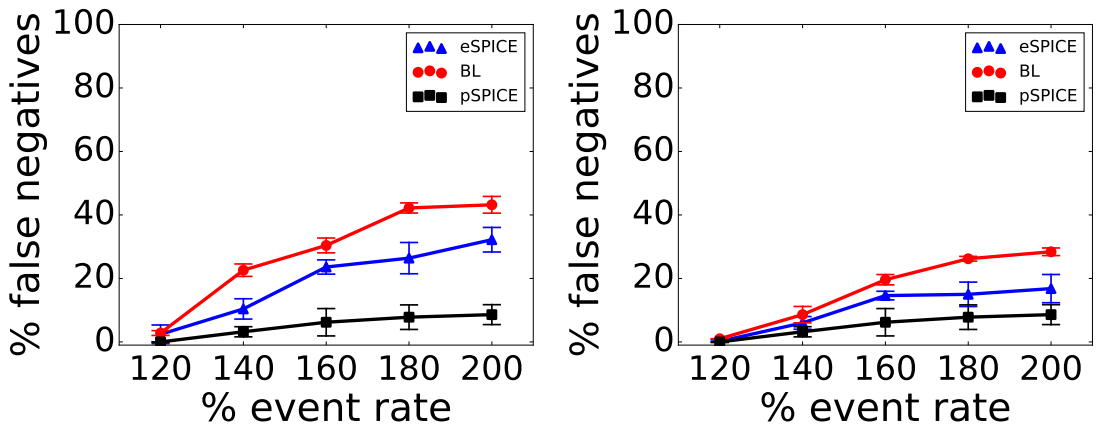


(a) strict QoR **(b)** relaxed QoR

Figure 4.9: False negatives for Q_5 with different input event rates.

(any stock company). Hence, in the case of Q_5, the majority of events in a window have similar utility values. Using relaxed QoR, eSPICE outperforms pSPICE as shown in Figure 4.9b. Figures 4.10a and 4.10b depict results for Q_6 when using strict and relaxed QoR, respectively. Both Figures 4.10a and 4.10b show that increasing the input event rate results in increasing the percentage of false negatives for all load shedders. The performance of eSPICE is better than the performance of BL when the input event rate is between 140% and 200%, however, it is worse than the performance of pSPICE in that range where the performance of pSPICE is considerably better in Q_6.

Number of false positives. Next, we show the impact of eSPICE on the number of false positives. Please note that in the case of relaxed QoR, performing load shedding might result in false positives only in the case of the negation operator where dropping negated events might result in false positives. For the *sequence* and the *any* operator dropping events cannot result in false positives in the case of relaxed QoR. Figures 4.11



(a) strict QoR **(b)** relaxed QoR

Figure 4.10: False negatives for Q_6 with different input event rates.

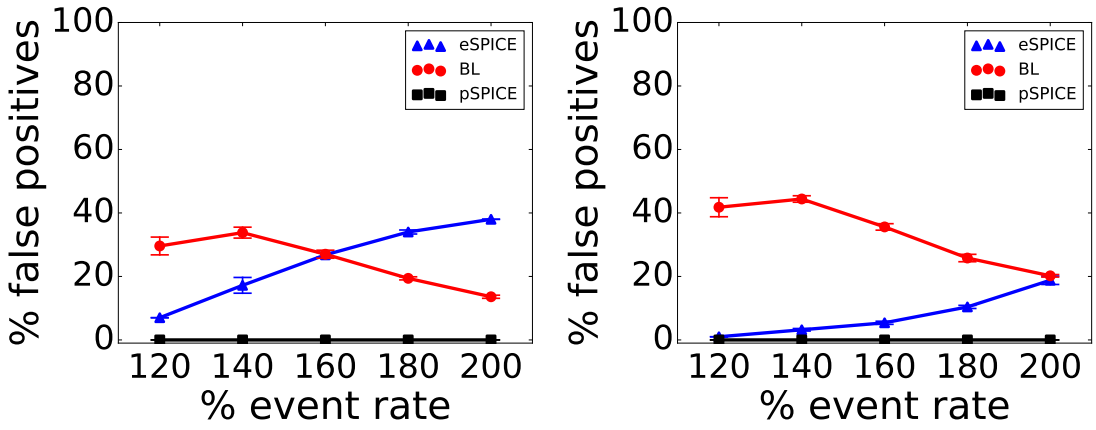
and 4.12 depict the impact of input event rates on the number of false positives for queries Q_1, Q_2, Q_5, and Q_6, respectively. We observed similar results for Q_3 and Q_4, hence we do not show them. In the figures, the x-axis represents the input event rate and the y-axis represents the percentage of false positives.

Figures 4.11a and 4.11b show results for Q_1 and Q_2, respectively. The percentage of false positives shown in Figure 4.11a caused by eSPICE increases when the input event rate increases. While the percentage of false positives caused by BL decreases when the input event rate increases. The reason for that is, when the event rate increases, more events are dropped from windows, hence it becomes hard to detect complex events. That results in fewer percentage of false positives. BL even results in less false positives than eSPICE when the input event rate is equal to or higher than 160%. Figure 4.11a shows that pSPICE results almost in no false positives. The results for Q_2 show similar behavior as depicted in Figure 4.11b. As a result, dropping PMs (i.e., using pSPICE) has a negligible impact on the false positives.

Figures 4.12a and 4.12b show results for Q_5 and Q_6, respectively. In Figure 4.12a, the percentage of false positives caused by eSPICE again increases when the input event rate increases. While the percentage of false positives caused by BL decreases when the input event rate increases. Here again, pSPICE has a negligible impact on the number of false positives. Figure 4.12b shows similar results for eSPICE and pSPICE. However, in the figure, the impact of BL on the percentage of false positives increases when the event rate increases.

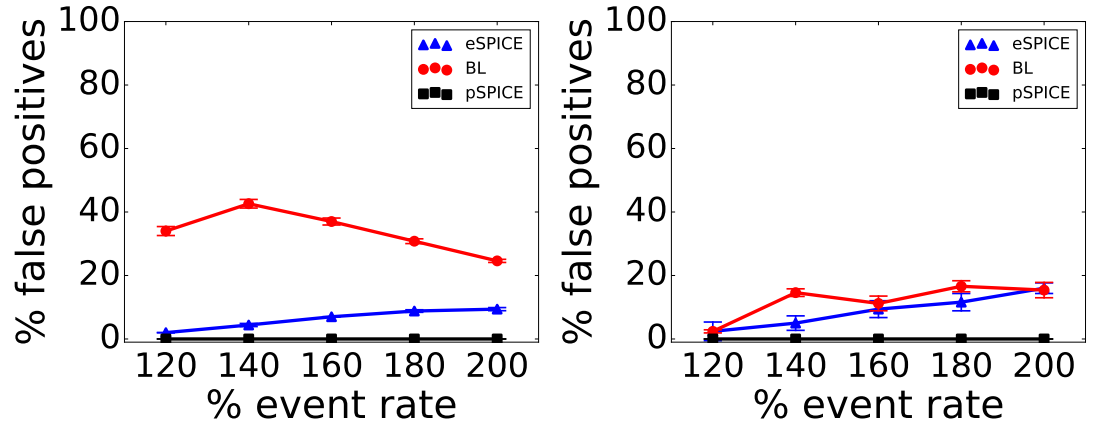
4.3.2.2 Impact of variable window size on QoR

Now, we show the impact of variable window size on the performance, w.r.t. QoR, of eSPICE. Using a 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



(a) strict QoR **(b)** strict QoR

Figure 4.11: False positives for Q_1 and Q_2 with different input event rates.



(a) strict QoR **(b)** strict QoR

Figure 4.12: False positives for Q_5 and Q_6 with different input event rates.

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 4.2.6.1. 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 Q_5 and Q_6 where we use several window sizes during model building to enforce having a different number of events per window.

For Q_5, we use a time-based sliding window of the following sizes: ws= 180, 200, 240, 260, and 300 seconds. The average observed window size is $\approx$ 2000 events, and hence we use $N = 2000$ to build UT. Here, the window size $ws = 240$ seconds contains around 2000 events ($\approx N$). Therefore, 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 a percentage value compared to the reference window size (i.e., $ws = 240$ seconds), and hence the

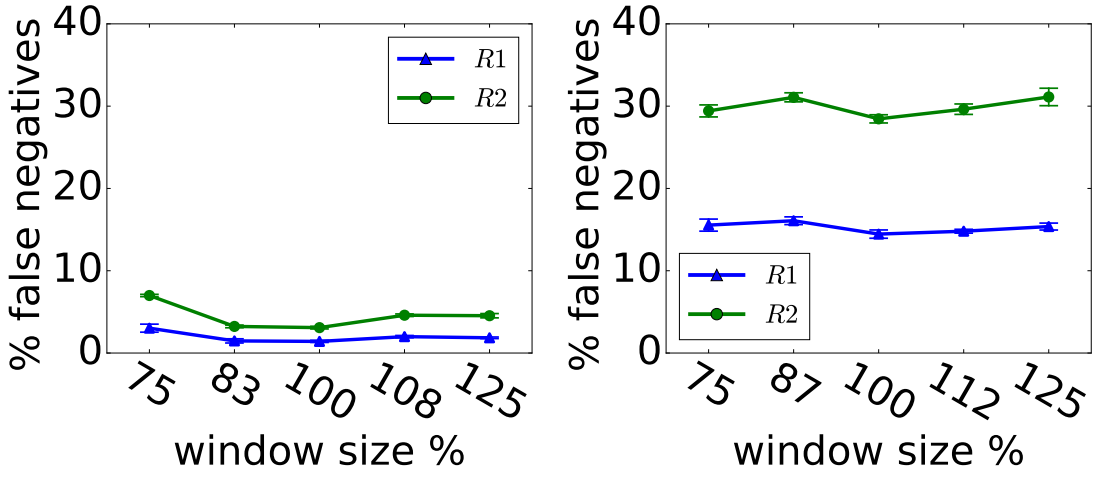
used windows are of the following sizes: 75%, 83%, 100%, 108%, and 125%. For Q_6, we again use a time-based sliding window of the following sizes: ws= 12, 14, 16, 18, and 20 seconds. The average observed window size is $\approx$ 800 events. Hence, we use $N = 800$ to build UT. As the window size $ws = 16$ seconds contains around 800 events ($\approx N$), 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 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 4.13 depicts the percentage of false negatives caused by eSPICE for both Q_5 and Q_6 using strict QoR. 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 4.13a shows results for Q_5 with two input event rates 120% (denoted by $R1$) and 140% (denoted by $R2$) while Figure 4.13b shows results for Q_6 with the two rates. Figure 4.13a shows that the percentage of false negatives increases when the difference between N and the window size increases regardless of the input event rate. While Figure 4.13b shows that the percentage of false negatives for Q_6 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 reason why the impact of eSPICE on the percentage of false negatives for Q_5 is higher than its impact for Q_6 is that Q_5 has a longer pattern size (i.e., 1 stock leading company + 20 stock companies) than Q_6 (i.e., 1 striker + 4 players) 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 Q_5 is higher than the number of event types that start a new match in Q_6 (only two strikers).

4.3.2.3 Impact of bin size on QoR

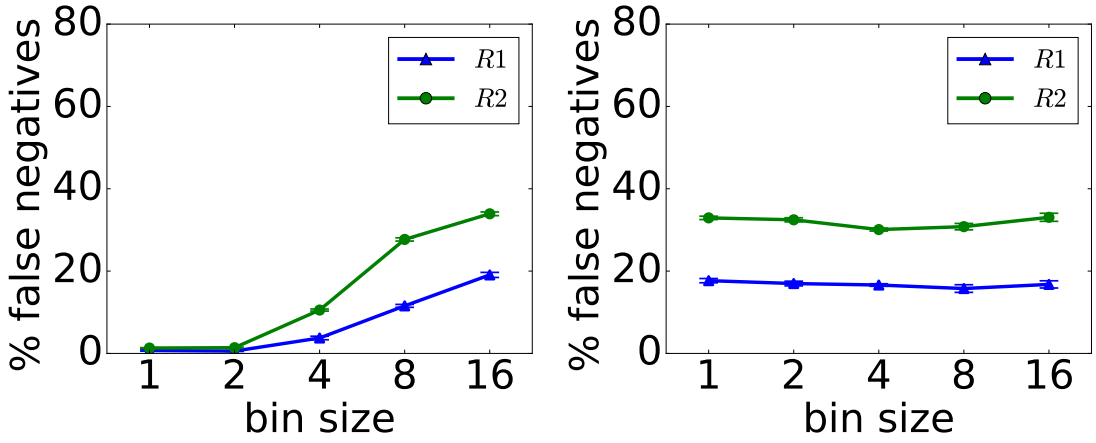
A big bin size might degrade QoR since it reduces the accuracy in UT of the important positions in the incoming windows. To analyze the impact of bin size on the performance, w.r.t. QoR, of eSPICE, we, again, run experiments with Q_5 and Q_6. We use a window of size $ws = 240$ seconds and $ws = 15$ seconds for Q_5 and Q_6, respectively. In addition, we use the following bin sizes for both queries: bs = 1, 2, 4, 8, 16.

Figure 4.14 depicts the percentage of false negatives for both queries with the strict QoR. The x-axis represents the bin size, and the y-axis represents the percentage of false negatives. Figure 4.14a depicts results for Q_5 with the input event rates $R1$ (120%)



(a) Q_5 **(b)** Q_6

Figure 4.13: Impact of variable window size on QoR.



(a) Q_5 **(b)** Q_6

Figure 4.14: Impact of bin size on QoR.

and $R2$ (140%) where it shows that the percentage of false negatives increases with the used bin size. Figure 4.14b depicts results for Q_6 with the input event 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 $R2$. The reason here is again similar to the reason in the variable window size experiment.

4.3.2.4 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

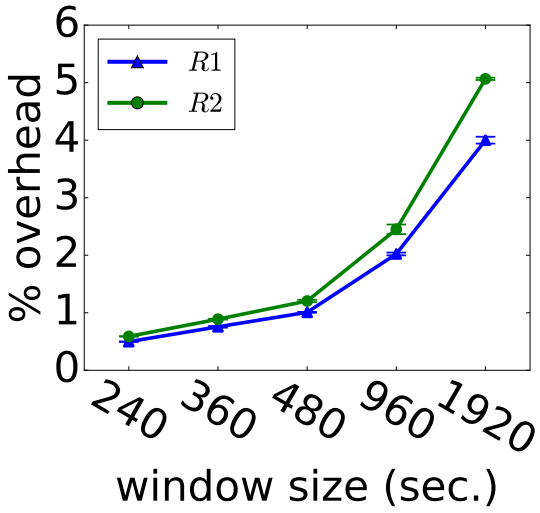


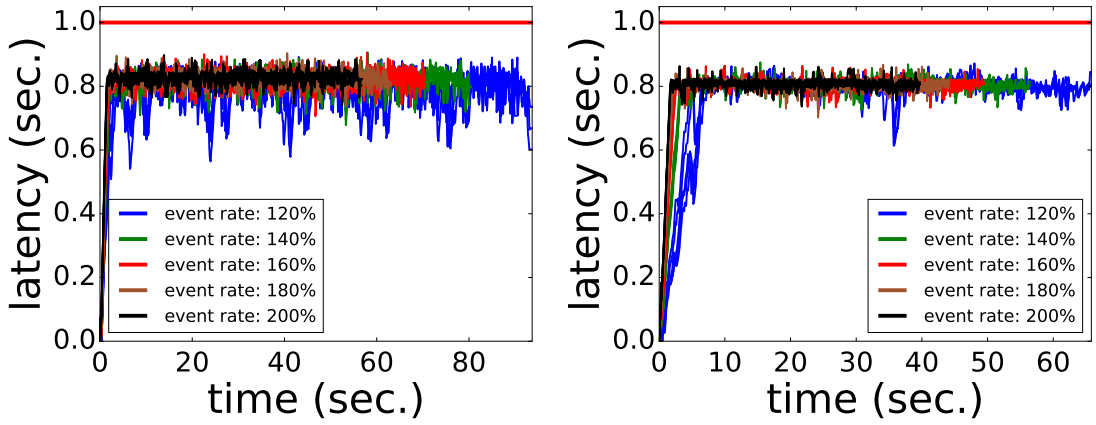
Figure 4.15: Q_5: Overhead of the LS.

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 Q_5 with two input event rate $R1$ (120%) and $R2$ (140%) 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 4.15 depicts the overhead of the LS for Q_5. The x-axis represents the used window size, and the y-axis represents the percentage 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 ($\approx$ 2000 events) to $\approx$ 5% with the window of size 1960 seconds ($\approx$ 16000 events). However, the overhead is still low compared to the actual event processing time. Hence, our load shedding strategy can maintain the given latency bound with low overhead. Moreover, the overhead of the window size can be reduced by increasing the bin size (bs). Additionally, improving the utility table locality in the memory can further reduce the overhead of LS.

4.3.2.5 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 4.16 shows the incurred event latency (l_e) when running Q_1 and Q_5 with different input event rates. 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 ($f * LB$) that is 800 milliseconds in this



(a) Q_1 **(b)** Q_5

Figure 4.16: Impact of bin size on the quality.

experiment.

4.3.2.6 Results Discussion

eSPICE performs much better than BL and pSPICE for the majority of queries. However, the performance of eSPICE varies for different classes of operators. QoR of eSPICE is exceptionally good for the *sequence* operators. The *sequence* operator ensures that every time only the same event types would match the pattern. that 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 event utilities are more sparse that adversely impacts the performance of eSPICE. Further, eSPICE shows its robustness against variable window size and bin size. The quality of results is only slightly influenced by a window size that is different from N or by a larger bin size. Moreover, the overhead of the LS component in eSPICE is very low compared to the actual processing overhead that makes eSPICE suitable for real-time complex event processing.

4.4 Conclusion

In this chapter, we proposed a lightweight load shedding approach, called eSPICE, for window-based CEP systems that 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 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, for the majority of queries, 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 the quality of results very low at minimum overhead.

Chapter 5

hSPICE: State-Aware Load Shedding from Input Event Streams

In the previous two chapters (Chapter 3 and 4), we presented our two proposed load shedding approaches pSPICE and eSPICE. pSPICE drops PMs from a window without considering the current events within the window, while eSPICE drops events from a window without considering the current open PMs within the window. As a result, pSPICE might drop PMs, that have relatively high utilities even if there exist events that may be dropped without impacting QoR. That might adversely impact QoR. On the other hand, eSPICE neither considers the importance nor the state of PMs. An event might have different utilities for individual PMs, depending on the importance and the state of these PMs. Therefore, combining these two load shedding approaches might overcome these drawbacks and result in a more powerful load shedder.

As a result, in this chapter, we propose a new *white-box* load shedding approach called hSPICE that combines the best of both pSPICE and eSPICE. In particular, hSPICE is a *white-box* load shedding approach that drops *events* either from *windows* or from *PMs*– it sheds events on window and PM granularities– while considering the operator's internal state. In hSPICE, events have different utilities for different PMs. 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. As we mentioned in Chapter 1, Section 1.2, 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. That 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 chapter 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.

The rest of the chapter is structured as follows. Section 5.1 presents the used system model. In Section 5.2, we explain in detail different components of hSPICE, the way the event utility is defined, how the event utility is predicted, and how load shedding is performed. Section 5.3 presents the obtained evaluation results. Finally, we conclude this chapter in Section 5.4.

5.1 System Model

In this chapter, we rely on a system model similar to the system model presented in Section 2.1, where we assume a window-based CEP system that consists of a one or more operators. An operator detects multiple patterns $\mathbb{Q}$ (i.e., multi-query). Each pattern $q_i \in \mathbb{Q}$ has a weight w_{q_i}, reflecting its importance. A pattern $q_i \in \mathbb{Q}$ is modeled as a finite state machine. In this chapter, we assume that 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., $j = \sum_{l=1}^{i-1} m_l$. For example, in Chapter 2, Example 1, pattern $q = seq(A; B; C)$ has four states (i.e., $m_i = 4$) where $\mathbb{S}_q = \{s_0, s_1, s_2, s_3\}$ as shown in Figure 2.3(a). The state s_0 represents the initial state of pattern q, and the state 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 Chapter 2, Example 1, since there is only one pattern (i.e., $\mathbb{Q} = \{q\}$), $\mathbb{S}_{\mathbb{Q}} = \mathbb{S}_q = \{s_0, s_1, s_2, s_3\}$.

A 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 been completed and become a complex event.

Therefore, in this chapter, the set of all possible states ($\mathbb{S}_\gamma$) of PM γ is defined as follows: $\mathbb{S}_\gamma = \mathbb{S}_{q_i} \setminus \{\textit{final states}\}$. Hence, the set of all possible states $\mathbb{S}_{\mathbb{\Gamma}}$ 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 Chapter 2, 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. 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 Chapter 2, Example 1, Figure 2.3(b), the sets 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 $\mathbb{\Gamma}_{w_3}^2 = \{\gamma_1, \gamma_2\}$.

In this chapter, we assume a *white-box* CEP operator. The operator reveals information about PMs and their progress (i.e., states) when processing primitive events within windows. Moreover, we assume that the set of event types ($\mathbb{T}$) in the input event stream is known. Additionally, we assume that the finite state machine is used as a computational model to detect patterns. However, hSPICE supports other CEP computational models. The discussion on supporting different computational models is similar to the discussion presented in Section 3.2.7. Therefore, we will not discuss supporting different computational models in this chapter.

5.2 hSPICE

The architecture of hSPICE is similar to the architecture of eSPICE, where we add three components to a CEP operator to enable load shedding: overload detector, load shedder (LS), and model (cf. Figure 5.1). However, the operator in hSPICE is a white-box operator, where the load shedder has access to the PMs.

Upon overload, 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 Chapter 4, Section 4.2.2.2. Our approach works with any drop interval. However, in this work, 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.

During overload, to maintain the given latency bound (LB), hSPICE drops input events that have the lowest adverse impact on QoR. 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

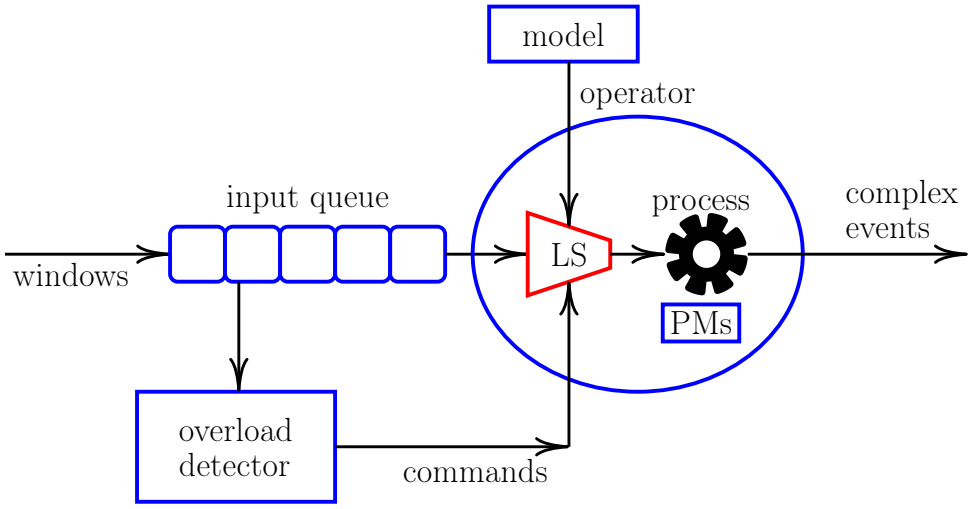


Figure 5.1: The hSPICE Architecture.

granularity) or from PMs within windows (referred to as partial match granularity). Dropping an event e from a PM γ means that event e is not processed (i.e., matched) with PM γ. 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. Recall that 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 in Section 2.1, an event in a window is processed (i.e., matched) 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 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.

5.2.1 Partial Match Granularity

5.2.1.1 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 importance. 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. The higher is the importance of γ in the window and the higher is the importance of event e for γ, the 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. That 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: $P(e \in \gamma \cap \gamma \textit{ completes})$, i.e., the probability that PM γ completes and event e contributes to PM γ. In window w, to predict $P(e \in \gamma \cap \gamma \textit{ completes})$ and hence $U_{e,\gamma}$, hSPICE uses three features: 1) current state S_γ of PM γ, 2) event type $T_e \in \mathbb{T}$, 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 5.1:

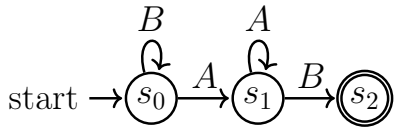
$$U_{e,\gamma} = f(T_e, P_e, S_\gamma) = w_{q_i} * P(e \in \gamma \cap \gamma \textit{ completes}) \tag{5.1}$$

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 (Section 2.1), 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. That is because, in the case of many remaining events in a window, a PM has a chance to be processed with more events than in the 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 Chapter 4. That 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., a certain position(s) within the window. In Example 1 (Section 2.1), in a 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 the 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.

5.2.1.2 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., $P(e \in \gamma \cap \gamma \textit{ completes})$, hence predicting the value of utility function $f(T_e, P_e, S_\gamma)$ in Equation 5.1. For ease of presentation, we introduce a simple running example which is depicted in Figures 5.2 and 5.3.



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 5.1: 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 5.2: Contribution ob_e and completion ob_γ observations.

Figure 5.2: 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

Figure 5.3: Computing event utility $U_{e,\gamma}$ for a partial match.

Example. 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 size is 5 events (i.e., $ws = 5$) and there are only two event types in the input event stream: A and B, i.e., $\mathbb{T} = \{A, B\}$.

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, 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., $\mathbb{\Gamma}_w^P$, (here P is the last position in w) are considered as *not completed* PMs.

Figure 5.2 shows an example of gathered observations on six PMs. Table 5.1 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 5.2 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 5.1, 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 5.2, 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 5.1).

Utility Prediction. hSPICE uses the gathered observations of both types (contribution ob_e and completion ob_γ) to predict the probability value $P(e \in \gamma \cap \gamma \textit{ completes})$, 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:

$$U_{e,s} = \frac{|\{e : e \in \gamma_s \ \&\ \gamma \textit{ completed}\}|}{|\{e : e \otimes \gamma_s\}|} \tag{5.2}$$

where $U_{e,s} = P(e \in \gamma_s \cap \gamma \textit{ completes})$. 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 5.3 shows the computed utility values $U_{e,s}$ from the observations shown in Table 5.2. 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 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 (i.e., $M = |\mathbb{T}|$), $N = \frac{ws}{bs}$, 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 $O(M.N.K)$. 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 $U_{e,\gamma}$. Another way is to monitor the distribution of events in the input event stream and rebuild the utility table whenever the event distribution changes by a certain threshold (cf. Chapter 4, Section 4.2.6.3).

5.2.1.3 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 5.1). 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. That 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.

5.2.1.4 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 5 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 5, 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 5, 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 5 Load shedder (PM granularity).

```
1: drop (T_e, P_e, S_γ) 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_γ) ≤ 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.

5.2.2 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. That 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. Recall 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.

5.2.2.1 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 5.3.

$$U_{e,w} = \sum_{\gamma \in \mathbb{\Gamma}_w^P} f(T_e, P_e, S_\gamma) \tag{5.3}$$

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:

$$U_{e,w} = \sum_{S_\gamma \in SM_w^P} f(T_e, P_e, S_\gamma) * SM_w^P(S_\gamma) \tag{5.4}$$

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 5.4 to compute the event utility $U_{e,w}$ in the window might considerably reduce the overhead of the utility computation. 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., $|\mathbb{S}_{\mathbb{\Gamma}}| < |\mathbb{\Gamma}_w^P|$ 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.

5.2.2.2 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 find a utility threshold u_{th} in a window that 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}.

5.2.2.3 Load Shedding

Now, we describe the way hSPICE drops events from windows. Algorithm 6 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 6, lines 2-3). On the other hand, if there is an 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 5.4 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 6, lines 4-9). That shows that hSPICE performs load shedding for window granularity in the worst case in a time-complexity of ($|\mathbb{\Gamma}_w^P|.O(1)$).

Algorithm 6 Load shedder (window granularity).

```
1:  applyLS (T_e, P_e, SM_w^P) begin
2:  if !isOverload then    // there is no overload hence no need to drop events
3:      return  False
4:  else
5:      compute U_{e,w} using Equation 5.4
6:      if U_{e,w} ≤ u_th then
7:          return  True
8:      else
9:          return  False

10: end function
```

5.3 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.

5.3.1 Experimental Setup

In this chapter, we use the same evaluation platform as in Section 3.3.1. We compare the performance, w.r.t. QoR, of hSPICE with the performance of eSPICE, pSPICE, and E-BL (cf. Section 3.3.1), where we rename E-BL to BL in this chapter. Moreover, we use the same datasets that are used in Section 4.3.1: 1) The *NYSE Stock Quotes* dataset that represents a stock quotes stream from the New York Stock Exchange. 2) The *RTLS* dataset that represents the position data stream from a real-time locating system in a soccer game. Additionally, we use the queries Q_1, Q_2, Q_3, Q_4, and Q_6 that are presented in Section 4.3.1. Moreover, we use the time-based sliding window strategy and the same selection and consumption policies used in Section 3.3.1.

5.3.2 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_6, we use a *time-based* sliding window and a *time-based* predicate. The number of defenders in Q_6 is 3 (i.e., $n = 3$). 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. The higher is the input event rate, the 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

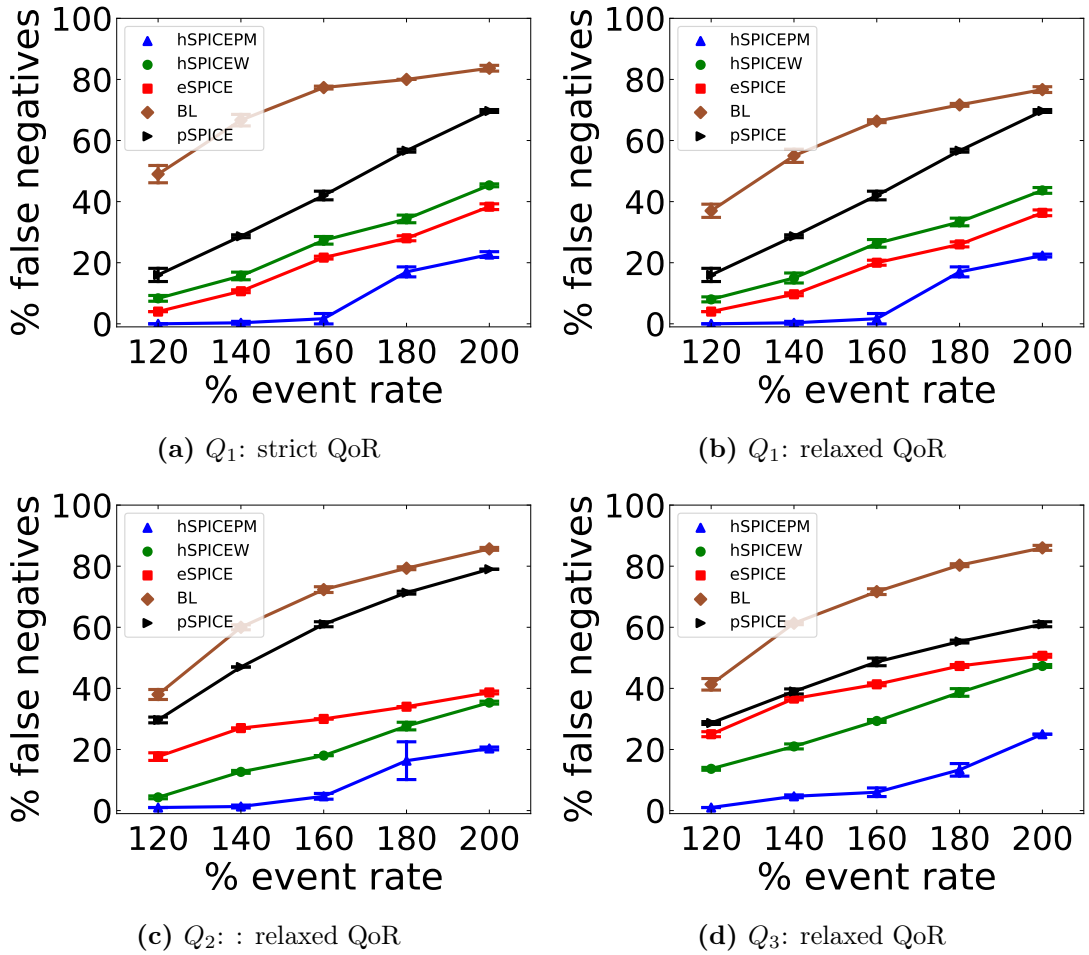
positives and false negatives for all queries (i.e., Q_1, Q_2, Q_3, Q_4, and Q_6). Additionally, when using relaxed QoR, applying load shedding might result in false negatives for all queries as well. However, it might result in false positives only in the case of Q_4 since Q_4 has a negation operator. If the negated event is dropped by the load shedder, it might result in a false positive.

5.3.2.1 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_6. 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_6, 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_6, 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_6 are as follows: 23K, 14K, 8K, 36K, 27K events/second, respectively.

Impact on False Negatives. Figures 5.4 and 5.5 depict the impact of event rates on false negatives for all queries. Figure 5.6 shows the ratio of dropped events or PMs (for pSPICE) with different event rates for Q_1 and Q_6. We observed similar results for Q_2, Q_3, and Q_4, hence we do not show them. In these figures, the x-axis represents the event rate. The y-axis in Figures 5.4 and 5.5 represents the percentage of false negatives while, in Figure 5.6, it represents the ratio of dropped events/PMs. Please note that measuring the load shedding overhead in hSPICEPM is very costly since the shedding is performed on the finest granularity. Therefore, in this Section, unlike Sections 3.3 and 4.3, instead of measuring the load shedding overhead directly, we measure the drop ratio that gives an indication of the load shedding overhead and can be measured with low overhead.

The percentage of false negatives might increase if the input event rate increases since more events/PMs must be dropped. Figure 5.4a and Figure 5.6a show the percentage of false negatives using strict QoR and the percentage of drop ratio for Q_1, respectively. As shown in Figure 5.4a, 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 5.6a. 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 5.6a 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



(a) Q_1: strict QoR

(b) Q_1: relaxed QoR

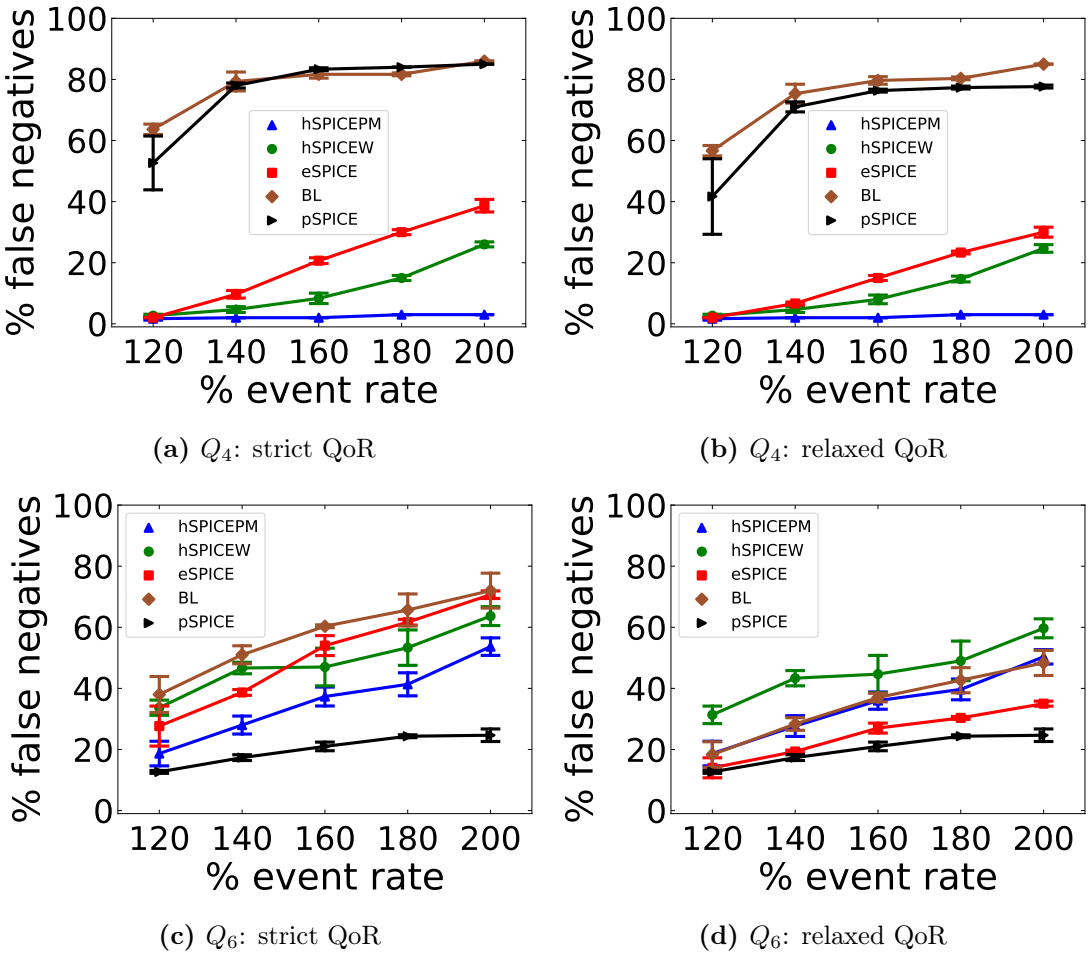
(c) Q_2: : relaxed QoR

(d) Q_3: relaxed QoR

Figure 5.4: Impact of event rate on false negatives for Q_1, Q_2, and Q_3.

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 5.4a, 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 5.6a. 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 is much higher, which might be used in some applications, hSPICEW may perform better than hSPICEPM as we will show in Section 5.3.2.2. The performance of hSPICEW is also worse than the performance of



(a) Q_4: strict QoR

(b) Q_4: relaxed QoR

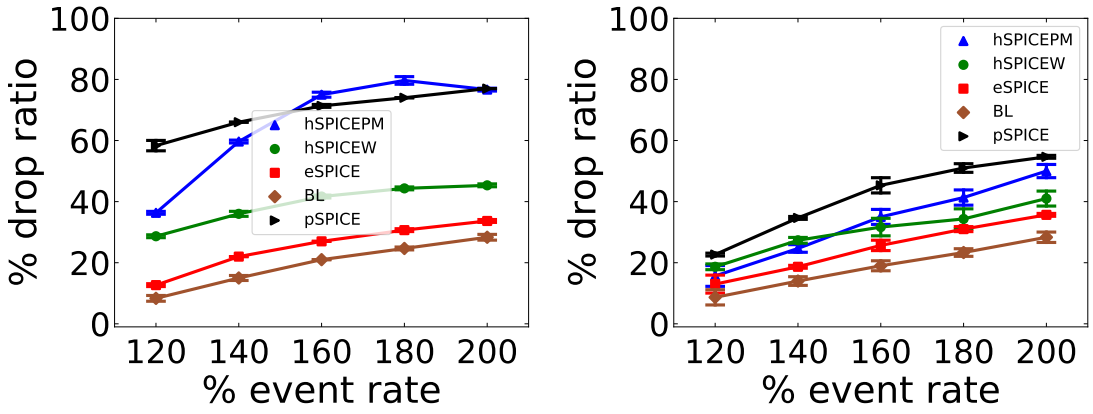
(c) Q_6: strict QoR

(d) Q_6: relaxed QoR

Figure 5.5: Impact of event rate on false negatives for Q_4 and Q_6.

eSPICE as shown in Figure 5.4a. This is because hSPICEW drops more events than eSPICE as depicted in Figure 5.6a 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 5.4b.

Figures 5.4c 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



(a) Q_1 **(b)** Q_6

Figure 5.6: Impact of event rate on drop ratio.

in Figure 5.4d. 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.

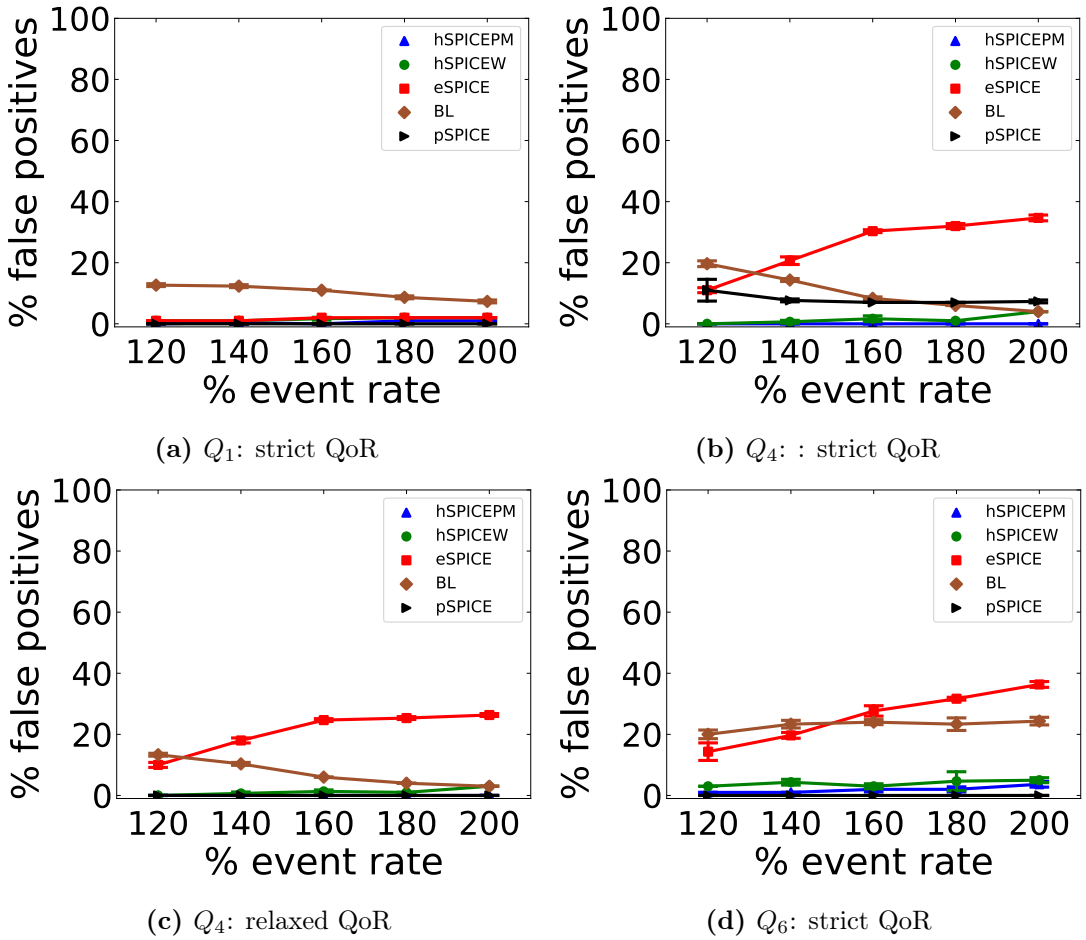
Figure 5.5a and 5.5b depict the percentage of false negatives for Q_4 (sequence with negation operator) using strict and relaxed QoR, respectively. In Q_4, we limit the number of complex events to only one event per window, where the window is closed if a complex event is detected. We do that to determine 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 5.5a and 5.5b 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 5.5c and 5.5d show the percentage of false negatives for Q_6 (sequence with any operator) using strict and relaxed QoR, respectively. While Figure 5.6b shows the ratio of dropped events/PMs for Q_6. The drop ratio in Figure 5.6b increases when the event rate increases. However, the drop ratio of hSPICEPM and hSPICEW for Q_6 is lower than their drop ratio for Q_1. This is because the cost of processing events in Q_6 is higher than the cost of processing events in Q_1. Therefore, in Q_6, the overhead of performing load shedding in comparison to the event processing cost is lower which results in a low drop ratio. In Figures 5.5c and 5.5d, the percentage of false negatives caused by all load shedders increases when the input event rate increases.

Figure 5.5c shows that the performance of hSPICEPM is better than the performance of hSPICEW, eSPICE, and BL. However, pSPICE outperforms hSPICEPM. However, Figure 5.5d (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 5.2.2. 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_6. This is because in Q_6 the utility values are spread and less accurately predicted since Q_6 represents an *any* operator in comparison to other queries that use a *sequence* operator. Q_6 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_6, 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 5.7 depicts the percentage of false positives with different event rates for queries Q_1, Q_4, and Q_6. 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 5.7 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 5.7a. While the percentage of false positives caused by eSPICE increases with increasing the event rate for Q_4 and Q_6. Figure 5.7 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, the higher is the probability



(a) Q_1: strict QoR

(b) Q_4: : strict QoR

(c) Q_4: relaxed QoR

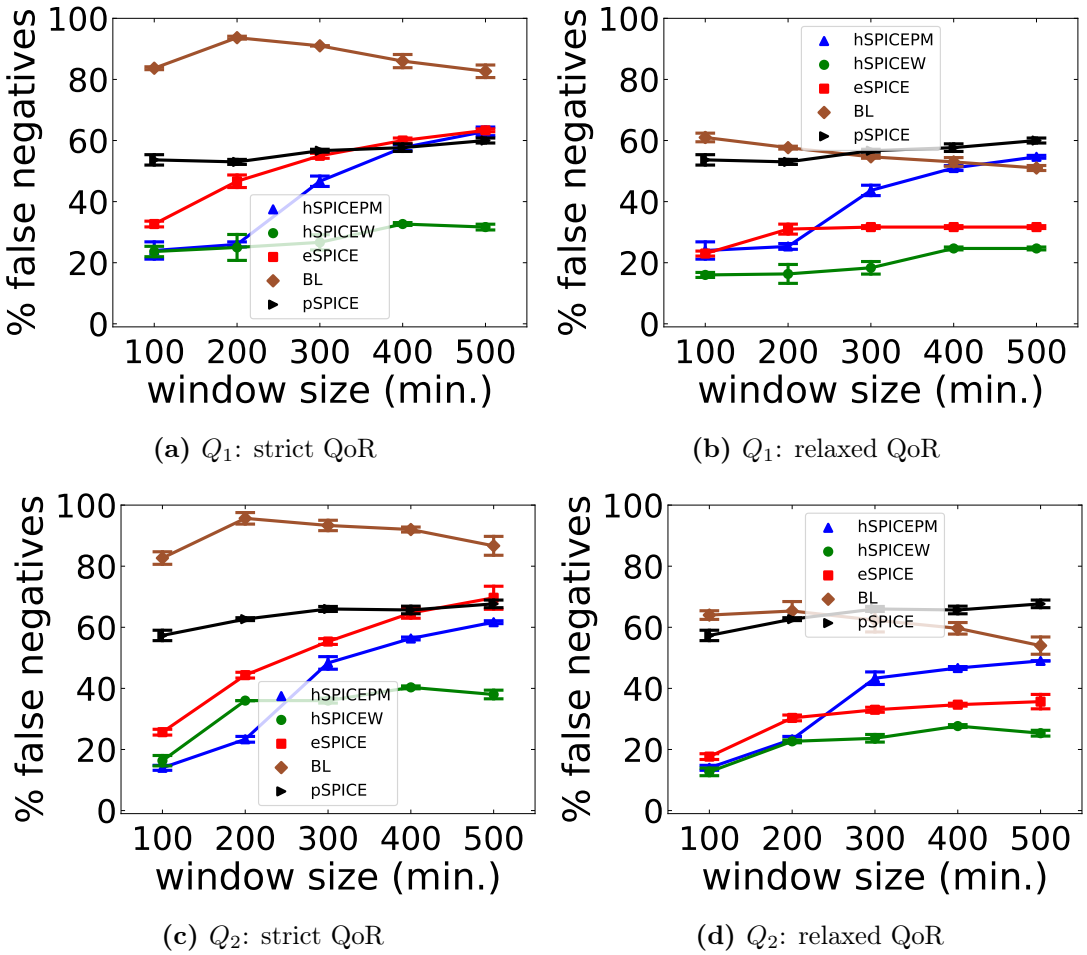
(d) Q_6: strict QoR

Figure 5.7: Impact of event rate on false positives.

to get false positives. The percentage of false positives caused by pSPICE in the case of Q_1, Q_4 (using relaxed QoR), and Q_6, is negligible as depicted in Figures 5.7a and 5.7d. While the percentage of false positives caused by pSPICE slightly decreases with increasing the event rate for Q_4, using strict QoR.

5.3.2.2 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 a



(a) Q_1: strict QoR

(b) Q_1: relaxed QoR

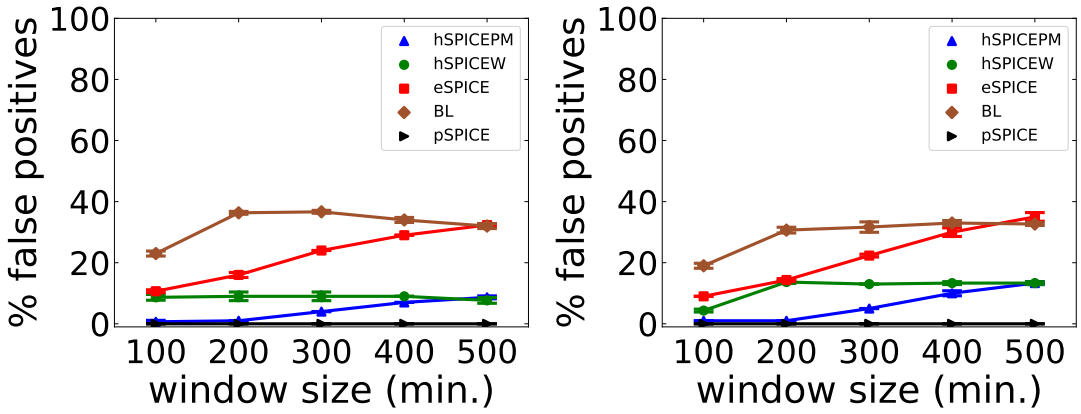
(c) Q_2: strict QoR

(d) Q_2: relaxed QoR

Figure 5.8: Impact of window size on false negatives.

few lookups in UT compared to hSPICEPM since hSPICEW performs a lookup in UT only once for each distinct PM state (cf. Section 5.2.2). That 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 5.2. 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 5.8 and Figure 5.9 depict the results for both queries. In both figures, the x-axis represents the event rate. The y-axis in Figure



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

Figure 5.9: Impact of window size on false positives.

5.8 represents the percentage of false negatives while the y-axis in Figure 5.9 represents the percentage of positives. We observed similar results for Q_3, Q_4, and Q_6, hence we do not show them.

Figure 5.8a 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 the window size might result in increasing the completion probability of PMs within the window. That 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. That 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. That is observed in Figure 5.8a, 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 show that the percentage of false negatives is only slightly increased when increasing the window size to 200 minutes after that it starts to decrease. This shows that hSPICEW performs,

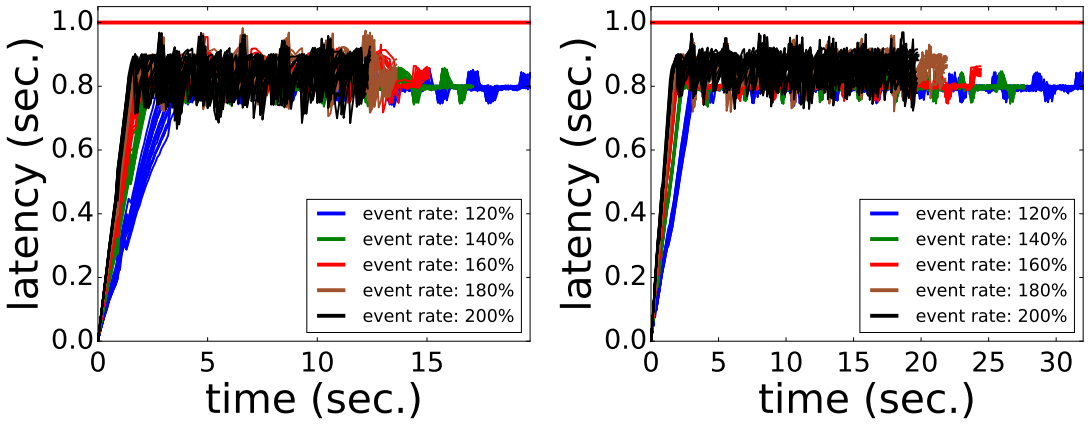
w.r.t. the percentage of false negatives, very well with relatively large window sizes and it outperforms eSPICE, BL, and pSPICE regardless of the used window size.

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 5.8b. 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 5.8c and 5.8d where hSPICEW performs very well regardless of the used window size. Figures 5.9a and 5.9b 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.

5.3.2.3 Maintaining Latency Bound

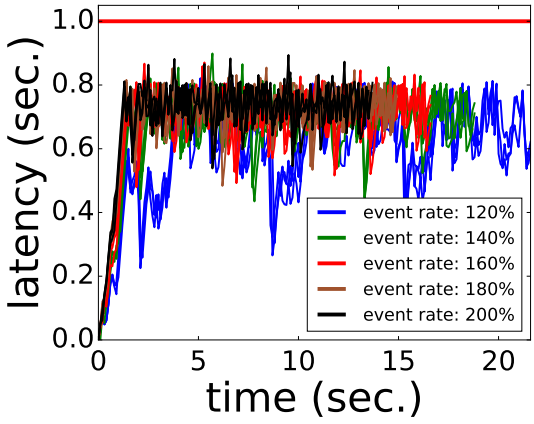
The main objective of hSPICE is to minimize the degradation in QoR while maintaining a given latency bound (LB). As mentioned above, LB is 1 second, and hSPICE drops events when the event queuing latency is greater than or equal to 80% of LB (i.e., 800 milliseconds). The event rate is an important factor 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 5.10 shows the event latency for Q_1, Q_2, and Q_6 where the event latency is the sum of the event queuing latency and the event processing latency. The event latency depicted in Figure 5.10 is measured when evaluating those three queries using the same settings as in Section 5.3.2.1. In the figure, the x-axis represents the elapsed time 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 5.10a, 5.10b, and 5.10c depict results for Q_1, Q_2, and Q_6, 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.



(a) Q_1

(b) Q_2



(c) Q_6

Figure 5.10: Maintaining latency bound.

5.3.2.4 Discussion

hSPICE shows its ability to maintain the given latency bound while reducing 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.

5.4 Conclusion

In this chapter, we proposed an efficient, lightweight load shedding strategy called hSPICE that combines the advantages of both eSPICE and pSPICE. 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.

Chapter 6

gSPICE: Generic Feature-Based Event Shedding

In the previous chapters, we proposed two white-box (i.e., pSPICE and hSPICE) and one black-box (i.e., eSPICE) load shedding approaches. In the white-box approach, the load shedder has access to the operator's internal state, i.e., PMs. While in the black-box approach, the operator's internal state is not visible. A clear advantage of a black-box shedding approach is that the shedding functionality can be easily added to CEP operators with minimal overhead on a domain expert. In fact, there is no need to modify the operator implementation. As a result, such a load shedder, that performs shedding agnostic to the operator implementation, has a universal appeal. Therefore, in this chapter, we focus on a black-box shedding approach that performs load shedding by dropping events. Using a black-box shedding approach, only dropping events is possible since the operator's internal state is not revealed, i.e., the load shedder does not have access to PMs.

So far, to predict the utility of events, we have used a limited set of features, such as the event type and the event position within the window. However, other important features may exist that help in accurately predicting the utility of an event. As a result, in this chapter, we propose to explore a new set of features that not only considers features such as event type but also the following two features: predecessor pane and event content. We define the predecessor pane of event e as the sequence of events that occur before event e in the input event stream. We consider the predecessor pane an important feature since it indicates the current progress of PMs within the operator. We also consider the event content an important feature since, in CEP, events in patterns are correlated while fulfilling certain predicates on the event content (i.e., the event attributes).

As a result, in this chapter, we propose a novel *black-box* load shedding approach for CEP systems, called gSPICE. In overload cases, to maintain a given latency bound, gSPICE drops *events* either from *windows* or from the *input event stream* of a CEP oper-

ator, i.e., it sheds events on two granularities: the window and the stream granularities. To minimize the shedding impact on QoR, gSPICE drops events with the lowest utilities, where it uses a probabilistic model to predict event utilities. The probabilistic model in gSPICE depends on the following features to predict event utilities: the predecessor pane, event content, event type, and event position within the window (when dropping events on the window granularity).

Using complex features such as the predecessor pane and event content to predict the event utility, on the one hand, might improve the prediction accuracy. But on the other hand, it might result in a heavyweight model that consumes high computational time to predict event utilities. gSPICE predicts event utilities with good accuracy while keeping the shedding overhead considerably low.

In particular, our contributions in this chapter are as follows:

- We propose gSPICE, a black-box load shedding approach for CEP systems, that, in overload cases, drops events either from windows or from the input event stream of a CEP operator (i.e., it sheds events on the window and the stream granularities) to maintain a given latency bound. gSPICE uses a probabilistic model to predict event utilities depending on the following features: 1) event type, 2) predecessor pane, 3) event content, and 4) event position within the window (when shedding events on the window granularity).

- We develop a data structure that depends on the Zobrist hashing [Zob90] to efficiently store the event utilities. This data structure enables gSPICE to perform load shedding in a lightweight manner.

- We also propose to use well-known machine learning approaches, e.g., decision trees or random forests, to estimate event utilities.

- We perform extensive evaluations on several real-world and synthetic datasets and a representative set of CEP queries to show the performance of gSPICE, w.r.t. its impact on QoR, and compare its performance with state-of-the-art shedding approaches.

The rest of this chapter is structured as follows. Section 6.1 presents the used system model. In Section 6.2, we explain in detail how gSPICE predicts the event utilities and performs load shedding. Section 6.3 presents the obtained evaluation results. Finally, we conclude this chapter in Section 6.4.

6.1 System Model

In this chapter, we rely on a system model similar to the system model presented in Section 2.1. We assume a window-based CEP system consisting of one or more operators. An operator detects multiple patterns $\mathbb{Q}$ (i.e., multi-query). Each pattern $q_i \in \mathbb{Q}$

has a weight w_{q_i}, reflecting its importance. In gSPICE, we extend the system model presented in Section 2.1 by introducing the predecessor pane and the type frequency in the predecessor pane. To clarify the system model, let us first introduce the following example.

Example 1. In a retail management application, every item is equipped with an RFID tag where there are three primitive events that may be generated for each item by RFID readers [WDR06]: 1) a shelf reading event (R) if an item is removed from a shelf, 2) a counter reading event (C) if the item is checked out on the counter, 3) an exit reading event (X) if the item is carried outside the retail store. To detect shoplifting, a CEP operator matches a pattern q which correlates events generated by RFID readers. Pattern q is defined as follows: generate a complex event if there exists a shelf reading event R and an exit reading event X for an item M but there is no counter reading event X for the item M within a certain time, e.g., two hours (i.e., the window length equals two hours). We may write this pattern as a sequence event operator with the negation event operator [CM94; WDR06]:

pattern **seq** $(R; !C; X)$
where $R.ID = C.ID$ **and** $R.ID = X.ID$
within 2 hours

In this example, the set of patterns $\mathbb{Q} = \{q\}$ and the event types represent the shelf reading R, counter reading C, and exit reading X, hence $\mathbb{T} = \{R, C, X\}$. Moreover, in this example, there is only one event attribute which is the item ID, hence the set of event attributes $\mathbb{E}_e = \{ID\}$. Let us assume an event input stream S_{in} with the following events, as depicted in Figure 6.1: $X_4\ R_3\ X_2\ C_1\ R_0$, where event E_i is of type E at position i in the input event stream, i.e., i defines the event order in the input event stream S_{in}. Assume that a window w contains the events $X_4\ R_3\ X_2\ C_1\ R_0$. Processing events in window w to detect pattern q is performed as follows: Processing event R_0 opens a PM γ_1, which is abandoned when processing event C_1 in the window w since the event type C is a negated event in pattern q, assuming that $R_0.ID = C_1.ID$. Events R_0 and C_1 *contribute* to PM γ_1 (i.e., $R_0, C_1 \in \gamma_1$), as events R_0 and C_1 update the progress (i.e., the state) of PM γ_1. Processing event X_2 in window w does not result in any match since there exists no open PMs, hence event X_2 does not contribute to any PM. The event R_3 opens a new PM γ_2 which completes and becomes a complex event when processing event X_4 in window w, assuming that $R_3.ID = X_4.ID$. This means that processing window w results in detecting only one complex event $cplx_{34} = (R_3, X_4)$ from events R_3 and X_4. We refer to R_3 and X_4 as the events that *contribute* to the complex event $cplx_{34}$. Moreover, we refer to $cplx_{34}$ as the complex event of pattern q, denoted by $cplx_{34} \subset q$.

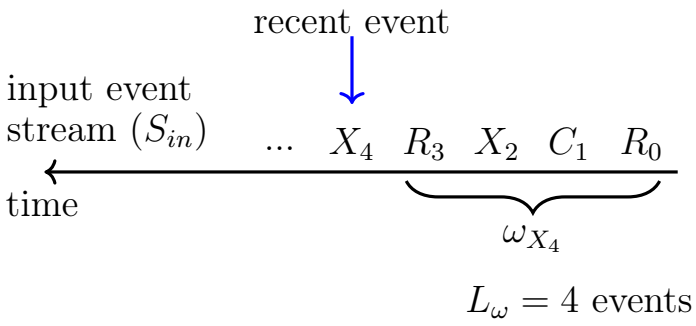


Figure 6.1: Predecessor pane.

6.1.1 Predecessor Pane

We now define the predecessor pane. The predecessor pane (denoted by ω_e) of an event e represents a *set* of a certain number of events that occur before event e in the input event stream S_{in}. The number of events in the predecessor pane is determined by the length of the predecessor pane (denoted by L_ω). The length of the predecessor pane might be either time-based or count-based. For example, a pane of length 5 seconds ($L_\omega = 5$ *seconds*), i.e., the predecessor pane ω_e of an event e contains all events within the last 5 seconds from event e. A pane of length 100 events ($L_\omega = 100$ *events*), i.e., the predecessor pane ω_e of an event e contains the last 100 events that occurred before event e in the input event stream S_{in}. Without loss of generality, to simplify the presentation, next, we will assume that the predecessor pane length L_ω is count-based if not otherwise stated. The predecessor pane ω_{e_j} of event e_j is formally defined as follows: $\omega_{e_j} = \{e_i : e_i \in S_{in}\ \&\ j - L_\omega \leq i < j\}$, where i and j represent the event order in the input event stream S_{in}.

Figure 6.1 depicts an example of input event stream S_{in} and the predecessor pane of event X_4. The pane is of length four, i.e., $L_\omega = 4$ events. In the figure, the predecessor pane ω_{X_4} contains events R_0, C_1, X_2, and R_3, i.e., $\omega_{X_4} = \{R_0, C_1, X_2, R_3\}$.

Type Frequency in Predecessor Pane. Next, we define the type frequency in the predecessor pane. Type Frequency (denoted by F_e) in the predecessor pane ω_e of an event e is a *sequence* representing the number of occurrences of each event type in the predecessor pane ω_e. Hence, $F_e = (F_{T_1}, F_{T_2}, ..., F_{T_m})$, where F_{T_i} represents the number of occurrences (i.e., frequency) of event type $T_i \in \mathbb{T}$ in the predecessor pane ω_e. More formally, for an event e, the type frequency F_e in the predecessor pane ω_e is defined as follows:

$$F_e = (F_{T_i} : \forall\ T_i \in \mathbb{T}, F_{T_i} = |\{e' : e' \in \omega_e \wedge T_i = T_{e'}\}|).$$

We may get the value F_{T_i} in type frequency F_e as follows: $F_{T_i} = F_e(T_i)$. In Figure 6.1, in the predecessor pane ω_{X_4}, there are two events of event type R, one event of type C, and one event of type X. Therefore, the type frequency in the predecessor pane ω_{X_4} is defined as follows: $F_{X_4} = (F_R, F_C, F_X) = (2, 1, 1)$, where $F_R = F_{X_4}(R) = 2, F_C =$

$F_{X_4}(C) = 1, F_X = F_{X_4}(X) = 1$.

gSPICE is a black-box load shedding approach, where the CEP operator only reveals the detected complex events. Additionally, gSPICE has access to events in the input event streams S_{in}, where set $\mathbb{T} = \{T_1, T_2, ..., T_m\}$ represents the set of all event types in the input event stream.

6.2 gSPICE

In this section, we present in detail our proposed load shedding approach that is called gSPICE. gSPICE has a similar architecture to eSPICE (cf. Section 4.2). Similarly to eSPICE, upon overload, in gSPICE, the overload detector computes the drop amount (denoted by ρ) and the drop interval. Then, it sends a drop command to the load shedder (LS) to drop ρ% from each drop interval to prevent the violation of the given latency bound (LB). LS drops events with the lowest utilities, where it gets the utilities from the model. In gSPICE, the computation of the drop amount ρ and drop interval is realized similarly to how they are realized in eSPICE (cf. Chapter 4). Therefore, in this chapter, we focus on predicting the event utilities and performing efficient event shedding.

gSPICE drops events either from windows or from the input event stream S_{in} of a CEP operator, i.e., it drops events on the window and stream granularities. Dropping events on the stream granularity implies that an event e is dropped from all windows to which the event e belongs. On the other hand, dropping events on the window granularity implies that events are dropped from windows individually, where an event might be dropped from a window, while it is processed in other windows. To minimize the negative impact of dropping events on QoR, gSPICE must drop those events that have the lowest utilities. Next, we first determine the features that are important to predict event utility in gSPICE. Then, we explain the way gSPICE predicts the event utilities. Finally, we show how load shedding in gSPICE is performed. As mentioned above, gSPICE drops events on two granularities, namely, the stream and window granularities, where we first focus on the stream granularity. Later, we discuss the needed modifications to perform shedding in gSPICE on the window granularity.

6.2.1 Event Utility

gSPICE depends on three features to predict utility (denoted by U_e) of an event e in the input event stream S_{in}: 1) event type $T_e \in \mathbb{T}$, 2) type frequency F_e in the predecessor pane ω_e, and 3) event attributes $\mathbb{E}_e$ (i.e., the event content). Event attributes are important features for predicting the event utility since a CEP pattern usually correlates events that contain attributes that fulfill specific conditions. In Section 6.1, Example 1, the pattern q matches events with the same ID (i.e., ID is an event attribute). Therefore, event attributes might have a considerable influence on the event utility. We consider only attributes with numerical values since more complex attributes (e.g., text or images)

might considerably increase the load shedding overhead, thus adversely impacting QoR. Event type T_e and type frequency F_e together represent important features to predict the event utility U_e as well. Type frequency F_e determines the importance of event e of type T_e since the type frequency (derived from the predecessor pane) contains information on events that happen before event e in the input event stream S_{in}. Thus, it gives an indication of the number of open PMs and the current progress of these PMs (i.e., states of these PMs). That, in turn, indicates the number of PMs to which event e of type T_e may contribute, hence the number of complex events to which event e might contribute.

For example, using the shoplifting query in Example 1 (cf. Section 6.1), Figure 6.2 depicts examples of two different predecessor panes of length $L_\omega = 4$ events for the event X_4. In this example, let us assume that there are at most four events before the event X_4 in windows that contain event X_4, i.e., event X_4 might contribute only to PMs that are opened by the latest four events before event X_4. In Figure 6.2(a), the predecessor pane of event X_4 contains four events of type R (i.e., R_0, R_1, R_2, R_3) where each event of type R opens a new PM. Since there are no events of type C before event X_4, event X_4 has a high probability to contribute to an open PM and results in detecting a complex event. Therefore, event X_4 should have a high utility. In Figure 6.2(b), there are three events of type R (i.e., R_0, R_1, R_2) and one event of type C (i.e., C_3) in the predecessor pane of event X_4. In this figure, event C_3 might abandon an already open PM. As a result, if events C_3 and X_4 are generated for the same item M (i.e., $C_3.ID = X_4.ID$), event X_4 will not contribute to any PM since the PM that event X_4 might contribute to is abandoned by event C_3. Hence, in this case, event X_4 is not important, and its utility should be low. However, if events C_3 and X_4 are generated for different items (i.e., $C_3.ID \neq X_4.ID$), event X_4 will contribute to an open PM and results in detecting a complex event. Hence, X_4 should be assigned a high utility value. As a result, in the above example, if the number of events of type C increases, the utility of event X_4 might decrease.

Another reason for considering type frequency F_e to predict the event utility is the following. For an event e, the value of event attributes $\mathbb{E}_e$ might be influenced by the occurrence of other events before event e in the input event stream S_{in}, i.e., in the predecessor pane ω_e of event e. For example, in a stock market application, a change in the stock quote of company A may influence the stock quote of company B within a certain time, i.e., the occurrence of an event of type A within the predecessor pane ω_e of an even e of type B (i.e., $T_e = B$) might influence the attribute values of event e of type B. As event attributes have an important impact on determining the event utility that implies that the predecessor pane also has an important impact on determining the event utility.

As a result, we write the utility U_e of event e as a function (called utility function) of these three features (i.e., event type $T_e \in \mathbb{T}$, type frequency F_e in the predecessor pane

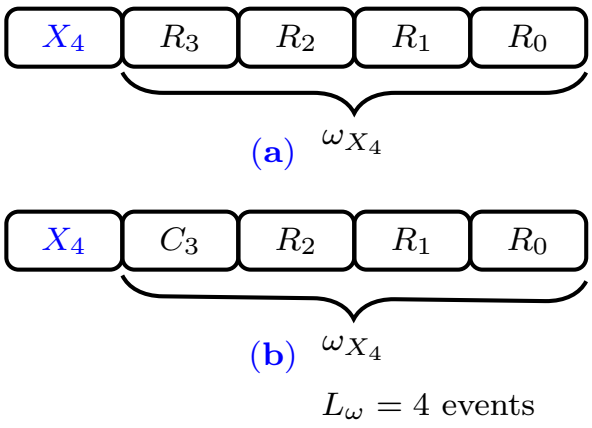


Figure 6.2: Importance of the predecessor pane.

ω_e, and event attributes $\mathbb{E}_e$) as shown in Equation 6.1:

$$U_e = f(T_e, F_e, \mathbb{E}_e) \tag{6.1}$$

6.2.2 Predicting Event Utility

Now, we explain how to predict the event utility U_e, i.e., build the utility function shown in Equation 6.1. gSPICE predicts the event utility depending on gathered statistics. Therefore, next, we first show how gSPICE gathers statistics and uses them to predict event utilities. Then, we explain the way gSPICE handles the predicted utilities. Let us first introduce the following simple examples.

Example 2. A CEP operator matches the pattern $q = seq(A; B)$. Assume that the input event stream contains only two event types A and B, and events have only a single attribute E_e, i.e., $\mathbb{E}_e = \{E_e\}$. Moreover, assume that a predecessor pane of length 3 events (i.e., $L_\omega = 3$) is used.

6.2.2.1 Gathering Statistics

To predict the value of the utility function f, gSPICE gathers statistics from the already processed events in the input event stream S_p. For each event $e \in S_p$, gSPICE builds an observation on the set of complex events (denoted by $\mathbb{M}_e$) to which event e contributes. The observation (denoted by ob_e) is of the following form: $ob_e\langle T_e, F_e, \mathbb{E}_e, \mathbb{M}_e\rangle$ where T_e is the type of event e, F_e represents the type frequency in the predecessor pane ω_e, and $\mathbb{E}_e$ is the set of attributes of event e. The observations ob_e are stored in a set called the observation set $\mathbb{S}_e$, i.e., $ob_e \in \mathbb{S}_e$. In Figure 6.3, Table 6.1 shows observations gathered for pattern q in Example 2. To simplify the presentation, the table shows observations only for event type A. In the table, F_A and F_B represent the frequency of event types A and B in the type frequency F_e in the predecessor pane ω_e, respectively, i.e., $F_A = F_e(A)$ and $F_B = F_e(B)$.

T_e	F_A	F_B	E_e	$\mathbb{M}_e$
A	1	2	5	$\{cplx_1\}$
A	1	2	5	$\{\}$
A	1	2	5	$\{cplx_2\}$
A	2	1	7	$\{\}$
A	2	1	7	$\{cplx_3\}$
A	2	1	8	$\{\}$
A	2	1	8	$\{\}$
A	2	1	8	$\{cplx_4\}$

Table 6.1: Observations $\mathbb{S}_e$.

T_e	F_A	F_B	E_e	M	O	U_e
A	1	2	5	2	3	0.67
A	2	1	7	1	2	0.50
A	2	1	8	3	1	0.33

Table 6.2: Aggregated Observations $\mathbb{S}_g$ and the predicted utilities.

Figure 6.3: Statistic gathering and utility calculation.

gSPICE aggregates observations that have the same values for event types T_e, type frequency F_e in the predecessor pane, and event attributes $\mathbb{E}_e$ into a set of aggregated observations (denoted by $\mathbb{S}_g$). An observation $ob_g \in \mathbb{S}_g$ is of the following form: $ob_g\langle T_e, F_e, \mathbb{E}_e, M, O\rangle \in \mathbb{S}_g$. M corresponds to the occurrences of complex events in the set $\mathbb{M}_e$ in observations $ob_e\langle T_e, F_e, \mathbb{E}_e, *\rangle \in \mathbb{S}_e$, where M represents the sum of the occurrences of the complex events in $\mathbb{M}_e$ multiplied by their weights, as complex events have weights reflecting their importance. O represents the number of occurrences of these observations $ob_e\langle T_e, F_e, \mathbb{E}_e, *\rangle \in \mathbb{S}_e$. The sign $*$ is used as a wildcard for the set of complex events $\mathbb{M}_e$ in the observations $ob_e \in \mathbb{S}_e$. The following equation formally formulates how gSPICE builds the aggregated observations.

$$
\begin{aligned}
&ob_g\langle T_e, F_e, \mathbb{E}_e, M, O\rangle \in \mathbb{S}_g : \\
&\qquad M = \sum_{ob_e\langle T_e, F_e, \mathbb{E}_e, \mathbb{M}_e\rangle \in \mathbb{S}_e} \; \sum_{cplx \in \mathbb{M}_e \;\&\; cplx \subset q_i} w_{q_i} \\
&\qquad O = |\{ob_e\langle T_e, F_e, \mathbb{E}_e, \mathbb{M}_e\rangle : ob_e\langle T_e, F_e, \mathbb{E}_e, \mathbb{M}_e\rangle \in \mathbb{S}_e\}|
\end{aligned}
\tag{6.2}
$$

In Figure 6.3, Table 6.2 shows the aggregated observations as a result of aggregating observations in Table 6.1. For example, in Table 6.1, an event e of type $T_e = A$ with a type frequency $F_e = (F_A, F_B) = (1, 2)$ and event attribute $E_e = 5$ occurs three times and contributes two times to a single complex event (i.e., $cplx_1$ and $cplx_2$). Hence, $O = 3$ and $M = 2$, where $cplx_1 \subset q$ and $cplx_2 \subset q$, assuming that the pattern's weight $w_q = 1$. That results in the following aggregated observation in Table 6.2: $ob_g\langle A, (1, 2), \{5\}, 2, 3\rangle$.

Please note that, as we mentioned above, if a query contains the negation event operator, a PM is abandoned when the negated event matches the pattern. Hence, in this case, no complex events are detected. Therefore, to capture the importance of the negated events, we assume that the CEP operator forwards the abandoned PMs to gSPICE to learn about the utility of these negated events.

6.2.2.2 Utility Prediction

After gathering statistics from η observations, gSPICE uses these observations to predict the utility function f, hence the event utility U_e. Equation 6.3 shows the way gSPICE computes the event utility U_e from the aggregated observations:

$$U_e = f(T_e, F_e, \mathbb{E}_e) = \frac{M}{O} : ob_g\langle T_e, F_e, \mathbb{E}_e, M, O\rangle \in \mathbb{S}_g \tag{6.3}$$

To compute the utility U_e of an event e of type T_e, with type frequency F_e, and event attributes $\mathbb{E}_e$, for an aggregated observation $ob_g\langle T_e, F_e, \mathbb{E}_e, M, O\rangle \in \mathbb{S}_g$, gSPICE divides M in the aggregated observation ob_g by the number of occurrences O in this aggregated observation ob_g. Table 6.2 also shows the computed utilities from the aggregated observations. For example, in the table, in the aggregated observation $ob_g\langle A, (1,2), \{5\}, 2, 3\rangle$, $M = 2$ and $O = 3$. Therefore, the utility U_e of an event e of type $T_e = A$ with a type frequency $F_e = (F_A, F_B) = (1,2)$ and event attribute $E_e = 5$ is calculated as follows: $U_e = \frac{2}{3} = 0.67$.

The distribution of events in the input event stream may change over time, where the predicted event utilities might become inaccurate. To keep the predicted event utilities accurate, the predicted utilities may be either periodically recompute or only when the distribution of events in the input event stream changes by a certain threshold.

To use the predicted event utilities U_e during load shedding, gSPICE handles the predicted utilities in the following two ways. 1) gSPICE stores the utilities in hash tables. We refer to this approach as gSPICE-SH. 2) gSPICE trains a well-known machine learning model (e.g., a decision tree or a random forest) with the predicted utilities to estimate the utility function. We refer to this approach as gSPICE-SM.

6.2.2.3 gSPICE-SH

In gSPICE-SH, we store the event utilities in a utility table (denoted by UT). The data structure used to store the utility table UT consists of hash tables. For each event type $T_e \in \mathbb{T}$, there is a hash table that stores the event utilities for all observed combinations of the type frequency F_e in the predecessor pane ω_e and event attributes $\mathbb{E}_e$. Hence, the utility of an event e of type T_e is stored in the utility table as follows: $U_e = f(T_e, F_e, \mathbb{E}_e) = UT[T_e][K]$, where the hash key K is computed from the type frequency F_e and event attributes $\mathbb{E}_e$. The value of event attributes $\mathbb{E}_e$ might occupy a wide range. Similarly, an event type $T_i \in \mathbb{T}$ might have a value between zero and L_ω in the type frequency F_e, i.e., $0 \leq F_e(T_i) \leq L_\omega$. That might result in a huge number of combinations of different event attributes $\mathbb{E}_e$ and type frequency F_e, especially if the length L_ω of the predecessor pane is large. That might considerably increase the required memory to store the utilities. To reduce the needed memory to store the utility table UT, we group the successive values of event attributes and the successive frequencies in the type frequency using bins of fixed sizes [KKP07]. To simplify the presentation,

we assume that a bin of size one is used for event attributes and type frequency if not otherwise stated.

To get the hash key K, hence the utility, we need to implement a hash function that combines the type frequency and event attributes. One way to implement the hash function is by using a function that iterates over event types in the type frequency F_e and over the event attributes $\mathbb{E}_e$ to compute the hash key K. The computational overhead of this approach depends on the number of event types in the type frequency and on the number of event attributes. The number of event types might be high, hence the computational overhead to get the key K might be high. To reduce the overhead of computing the hash key K, we use the Zobrist hashing [Zob90] as a hashing function to compute the key K.

Zobrist hashing depends on bitwise XOR operations (denoted by $\oplus$) to compute the hash key [Zob90]. To use the Zobrist hashing, we do the following. 1) We generate big unique random numbers for each possible frequency of an event type in the type frequency F_e. Hence, for each event type $T_i \in \mathbb{T}$, we generate the following set $\mathbb{R}_{T_i}$ of random numbers: $\mathbb{R}_{T_i} = \{R^l_{T_i} : 0 \leq l \leq L_\omega\}$ where $T_i \in \mathbb{T}$ and $R^l_{T_i}$ represents a big unique random number. Please note that an event type T_i might at most occur L_ω times in the type frequency F_e, i.e., $0 \leq F_e(T_i) \leq L_\omega$. 2) We also generate big unique random numbers for each possible range of each event attribute. Hence, for an event attribute $E_e \in \mathbb{E}_e$, we generate the following set $\mathbb{R}_{E_e}$ of random numbers: $\mathbb{R}_{E_e} = \{R^l_{E_e} : 0 \leq l \leq max(range)\}$ where $E_e \in \mathbb{E}_e$ and $R^l_{E_e}$ represents a big unique random number and $max(range)$ represents the maximum range an event attribute might have. 3) We use one hash key K_1 as a hash key for event types in the type frequency F_e. The hash key K_1 is computed as follows: $K_1 = \bigoplus R^{F_{T_i}}_{T_i} : \forall\ T_i \in \mathbb{T}, F_{T_i} = F_e(T_i), R^{F_{T_i}}_{T_i} \in \mathbb{R}_{T_i}$. The key K_1 is computed by performing XOR operations between the random numbers corresponding to the frequency of each event type in the type frequency. 4) We use a second hash key K_2 as a hash key for event attributes. The hash key K_2 is computed as follows: $K_2 = \bigoplus R^l_{E_e} : \forall\ E_e \in \mathbb{E}_e, l = range_E, R^l_{E_e} \in \mathbb{R}_{E_e}$, where $range_E$ represents the corresponding range for the value of the event attribute E_e. Similar to computing K_1, the key K_2 is computed by performing XOR operations between the random numbers corresponding to the range values of each event attribute. 5) The hash key K is computed by performing an XOR operation between the hash keys K_1 and K_2, i.e., $K = K_1 \oplus K_2$.

To reduce the overhead of computing the hash key K, we *continuously update* the hash key K_1. We first compute the hash key K_1 depending on all event types $T_i \in \mathbb{T}$ in the type frequency F_e. Then, when the predecessor pane changes, for all changed frequencies of event types in the type frequency, we remove the old values and add the new values to the hash key. Since XOR is a self-inverse operation (e.g., $X \oplus X = 0$), to remove an old value F'_{T_i} for the event type T_i in the type frequency F'_e, we need to perform only a single XOR operation. Additionally, we need a single XOR operation to add the new value F_{T_i} for the event type T_i in the type frequency F_e. Hence, for each changed

frequency of an event type, we need two XOR operations, i.e., $K_1 = K_1 \oplus R_{T_i}^{F'_{T_i}} \oplus R_{T_i}^{F_{T_i}}$, where $F'_{T_i} = F'_e(T_i), F_{T_i} = F_e(T_i)$ and $R_{T_i}^{F'_{T_i}}, R_{T_i}^{F_{T_i}} \in \mathbb{R}_{T_i}$. Therefore, if the predecessor pane changes by only one event (i.e., the predecessor pane shifts by one event), there is a need for four XOR operations at max to update the hash key K_1. That is because an event type might be removed from the predecessor pane (i.e., needs two XOR operations) and another event type might be added to the predecessor pane (i.e., needs two XOR operations). As a result, to compute a new hash key K, there is a need at most to (4 + $|\mathbb{E}_e|$) XOR operations. That represents a considerable reduction in the overhead of computing the hash key K, especially, in the case of a high number of event types.

6.2.2.4 gSPICE-SM

Another way to handle a huge number of predicted utilities is to use well-known machine learning models where we might use a machine learning model to estimate the utility function. In this case, the aggregated observations $\mathbb{S}_g$ are used as input training data to the machine learning model, and the corresponding computed utility values (cf. Equation 6.3) are used as labels. After training the model, the produced trained model represents the estimated utility function. Hence, to get the utility of an event e, gSPICE-SM provides the trained model with the event type T_e, type frequency F_e in the predecessor pane, and event attributes $\mathbb{E}_e$. The model returns the predicted utility value U_e.

Several machine learning models can be used to estimate the utility function, e.g., neural networks, decision trees, random forest, etc. However, machine learning models usually impose considerable computational overhead to predict the event utility that might not be tolerable for performing load shedding in CEP. Moreover, these models have many parameters that need to be tuned, which increases the burden on a domain expert. In gSPICE-SM, we use two machine learning models, namely, decision trees [Qui93; ASm09] and random forests [Tin95], to estimate the utility function. Please note that controlling the depth of trees in these models and other parameters, such as when to split nodes, can control the needed memory size by these two models. In Section 6.3, we show how good are these models in estimating the utility function and their imposed computational overhead.

6.2.3 Utility Threshold

As we mentioned above, in overload cases, gSPICE must drop ρ% of events during every drop interval (e.g., window) to maintain the given latency bound. To do that, gSPICE uses the predicted event utilities to find a utility threshold u_{th} that can be used to drop the required percentage of events ρ. That is done in a similar way to predicting utility threshold in Section 4.2.3. First, gSPICE uses the gathered statistics and the predicted event utilities to compute the percentage of occurrences (denoted by O_u) of an event utility u in the already processed event stream $\mathbb{S}_p$ in the gathered statistics, i.e.,

$O_u = \frac{|\{e : e \in \mathbb{S}_p, u = U_e\}|}{|\mathbb{S}_p|}$. Then, gSPICE accumulates the percentage of occurrences of event utilities O_u in ascending order to get the cumulative utility occurrences (denoted by O_u^c) as follows: $O_u^c = \sum_{u' \leq u} O_{u'}$. The cumulative utility occurrences O_u^c represents the percentage of events in the stream of already processed events $\mathbb{S}_p$ that have a utility value u' which is less than or equal to the utility u, i.e., $u \leq u'$. Hence, to drop ρ% of events from every drop interval, we may find the lowest cumulative utility occurrences O_u^c that is higher than or equal to ρ, and chooses the utility u as the utility threshold u_{th}, i.e., $u_{th} = u$: $O_u^c \geq \rho \; \forall \; O_{u'}^c \geq \rho \wedge O_u^c \leq O_{u'}^c$.

6.2.4 Load Shedding

In this section, we explain the way gSPICE uses the predicted event utilities and utility thresholds to drop events from the input event stream S_{in} during overload to maintain a given latency bound. Algorithm 7 formally defines how the load shedding is performed in gSPICE.

For each event e in the input event stream S_{in}, before event e is processed by the operator, gSPICE checks whether there is overload and a need to drop events. If there is no overload, the event is processed by the operator (cf. Algorithm 7, lines 2-3). Otherwise, the operator is overloaded and there is a need to drop events. In this case, gSPICE must drop ρ% of events from the input event stream S_{in} in every drop interval to maintain the given latency bound. Therefore, gSPICE first finds a utility threshold u_{th} that results in dropping ρ% of events using the cumulative utility occurrences as we explained above. In case the event utilities are stored using the Zobrist hashing, i.e., gSPICE-SH (cf. Algorithm 7, lines 4-10), gSPICE computes the hash key K by XORing the hash key K_1 for the type frequency F_e and the hash key K_2 for the event attributes $\mathbb{E}_e$ (cf. Algorithm 7, lines 5-6). Then, gSPICE gets the event utility U_e from the utility table UT and compares the utility U_e with the utility threshold u_{th}. If the event utility U_e is less than or equal to the utility threshold, the event e is dropped from the input event stream S_{in} (cf. Algorithm 7, lines 7-8). Otherwise, the event e is processed normally by the operator. In case a machine learning model is used to estimate the event utility, i.e., gSPICE-SM (cf. Algorithm 7, lines 11-15), gSPICE provides the model with the event type, type frequency, and event attributes. The model returns the predicted event utility U_e. Here again, if the event utility U_e is less than or equal to the utility threshold u_{th}, the event e is dropped from the input event stream S_{in} (cf. Algorithm 7, lines 12-13). Otherwise, event e is processed by the operator.

6.2.5 Window Granularity

So far, we have explained how to predict the event utilities and perform load shedding on the stream granularity. In this section, we show how to modify our proposed approach to be able to perform shedding on the window granularity. As mentioned above, an

Algorithm 7 Load shedder.

```
1:  drop (e) begin
2:  if !isOverloaded then // there is no overload hence no need to drop events.
3:      return False
4:  else if isZobrist then                                  // using gSPICE-SH.
5:      K_2 = ⨁ 𝔼_e        // computing K_2 by XORing the event attributes 𝔼_e.
6:      K = K_1 ⊕ K_2                                  // K_1 is continuously updated.
7:      if UT[T_e][K] ≤ u_th then
8:          return True
9:      else
10:         return False
11: else                                                    // using gSPICE-SM.
12:     if model.getUtility(T_e, F, 𝔼_e) ≤ u_th then
13:         return True
14:     else
15:         return False
16: end function
```

event in the input event stream might belong to several windows where windows may overlap. Therefore, dropping on the window granularity implies that events are dropped from each window individually, where an event might be dropped from a window while it is still there in other windows. Performing load shedding on the window granularity has the following advantage. The utility of events might be predicted in each window individually. Hence, the event utility might be predicted more accurately compared to predicting event utilities on the stream granularity since the window is more fine-grained than the stream. However, predicting the event utility in each window might impose a considerable overhead which might increase the negative impact of shedding on QoR.

As features to predict the utility U_e^w of event e in a window w, we use the same features that are used to predict the event utilities on the stream granularity, i.e., the event type T_e, type frequency F_e in predecessor pane ω_e, and event attributes $\mathbb{E}_e$. Additionally, we use one more feature which is the event position (denoted by P_e) within window w that represents an important feature since it captures the temporal correlation between events (cf. Chapter 4). As a result, we represent the event utility U_e^w in a window by the following event utility function:

$$U_e^w = f(T_e, F, \mathbb{E}_e, P_e) \tag{6.4}$$

On the window granularity, gSPICE similarly performs all the following tasks as

discussed on the stream granularity: predicting the event utilities using gathered statistics, handling predicted utilities using the Zobrist hashing (denoted by gSPICE-WH) or well-known machine learning models, e.g., decision trees and random forests, (denoted by gSPICE-WM), predicting the utility threshold, and performing load shedding. Therefore, we do not again explain them in this section. The window size might be large, hence the number of positions within the window might be large. This might increase the needed memory to store the utilities. To reduce the needed memory, here again, we use bins of fixed size that groups successive window positions.

6.3 Performance Evaluations

In this section, we evaluate the performance of gSPICE by using several datasets and a set of representative queries.

6.3.1 Experimental Setup

Here, we describe the evaluation platform, the baseline implementation, datasets, and queries used in the evaluations. In this chapter, we use the same evaluation platform as in Section 3.3.1. We compare the performance, w.r.t. QoR, of gSPICE with the performance of hSPICE, eSPICE, pSPICE, and E-BL (cf. Chapter 3, Section 3.3.1), where we rename E-BL to BL in this chapter. As hSPICE drops events either from windows, i.e., hSPICEW, or from PMs, i.e., hSPICEPM (cf. Chapter 5), in this section, we show the evaluation results for hSPICE by selecting the best results achieved by hSPICEW or hSPICEPM.

Datasets. The distribution of event types might considerably impact the performance of gSPICE. Therefore, to control the event distribution, we generate eight synthetic datasets as shown in Table 6.3, where events are generated using an exponential distribution. Datasets DS_1, DS_2, DS_3, and DS_4 contain events of three types: A, B, and C. While datasets DS_5, DS_6, DS_7, and DS_8 contain events of six types: A, B, C, D, E, and F. μ_X represents the average time (in seconds) between event instances of the event type X. μ_X controls the percentage of each event type in these datasets. In Table 6.3, we also show the average expected percentage (approximate values) of each event type in the datasets. Moreover, all events in all datasets have an attribute V_1 with uniformly distributed values between 1 and 10. We also use the two real-world datasets presented in Section 4.3.1. 1) The stock quote stream (NYSE dataset) from the New York Stock Exchange (NYSE). This dataset contains stock events that have a change in their quote by at least 0.4%. 2) The position data stream from a real-time locating system (RTLS dataset) in a soccer game.

Queries. We employ seven queries that cover an important set of operators in CEP as shown in Table 6.4: sequence, disjunction, sequence with Kleene closure, sequence with negation, and sequence with any operators [CM94; CM10; WDR06]. In the table,

	μ_A	μ_B	μ_C	μ_D	μ_E	μ_F	$A\%$	$B\%$	$C\%$	$D\%$	$E\%$	$F\%$
DS_1	2.5	15	40	-	-	-	81.3	13.6	5.1	-	-	-
DS_2	2.8	15	15	-	-	-	72.8	13.6	13.6	-	-	-
DS_3	4	6	12	-	-	-	50	33.3	16.7	-	-	-
DS_4	6	6	6	-	-	-	33.3	33.3	33.3	-	-	-
DS_5	2.5	15	40	2.5	15	40	40.7	6.8	2.5	40.7	6.8	2.5
DS_6	2.8	15	15	2.8	15	15	36.4	6.8	6.8	36.4	6.8	6.8
DS_7	4	6	12	4	6	12	25	16.7	8.3	25	16.7	8.3
DS_8	6	6	6	6	6	6	16.7	16.7	16.7	16.7	16.7	16.7

Table 6.3: Synthetic Datasets.

C_i represents the stock quote of company i, and D_i represents the event of player i. Queries Q_1, Q_2, and Q_3 are executed over the synthetic datasets. While queries Q_4, Q_5, and Q_6 are executed over the NYSE dataset and they are the same as queries Q_1, Q_2, and Q_4, respectively, presented in Section 4.3.1. Q_7 is executed over the RTLS dataset and it is the same as Q_6 presented in Section 4.3.1. We use the time-based sliding window strategy and the same selection and consumption policies used in Section 3.3.1.

6.3.2 Experimental Results

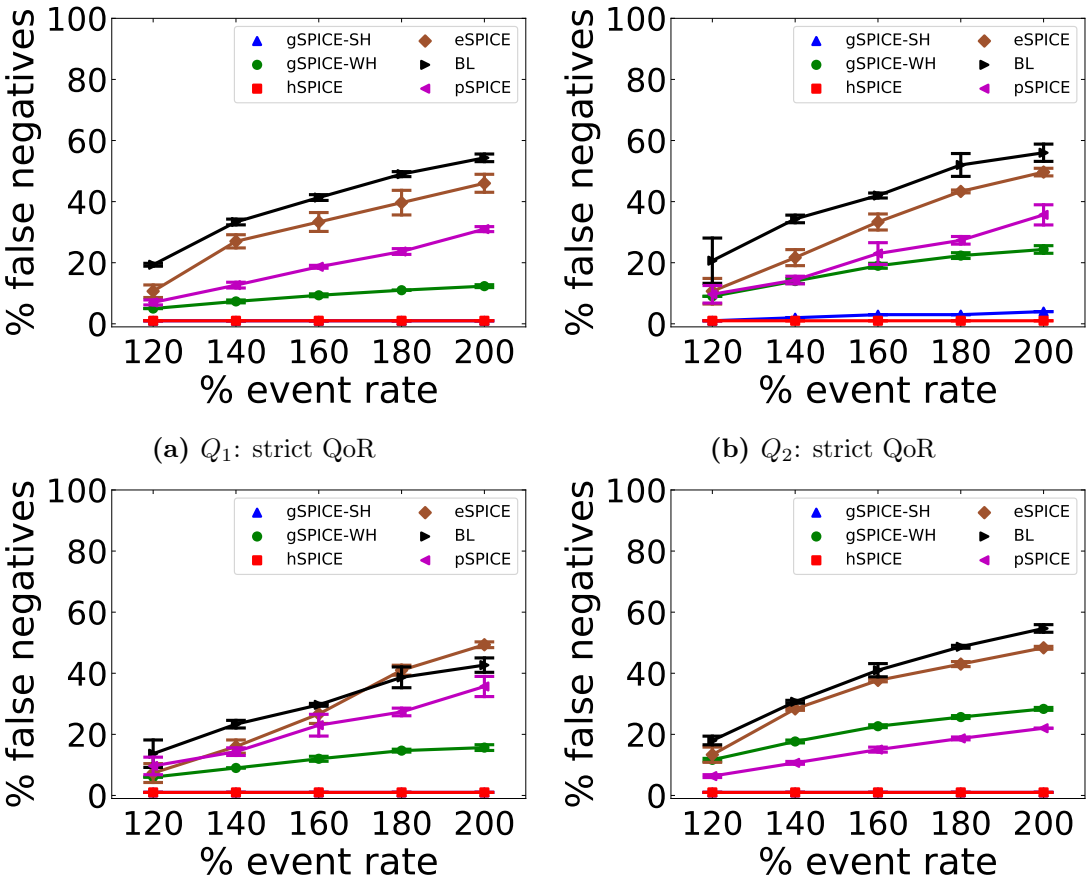
Next, we evaluate the impact of gSPICE on QoR, particularly on the number of false positives and false negatives, and compare its results with the results of hSPICE, eSPICE, pSPICE, and BL. Moreover, we show the overhead of gSPICE and its ability to maintain a given latency bound (LB). We first focus on the performance of gSPICE-SH and gSPICE-WH, i.e., when Zobrist hashing is used to store event utilities. Later, we also show the evaluation results of gSPICE when using a decision tree or a random forest.

If not noted otherwise, we employ the following settings. For all queries, we use a time-based sliding window and a time-based predicate. For queries based on synthetic data (i.e., Q_1, Q_2, and Q_3), we use a window of length 250 seconds. For Q_1 and Q_3, a new window is opened every 10 seconds (i.e., the slide size is 10 seconds). While for Q_2, a new window is opened every 20 seconds. For stock queries (i.e., Q_4, Q_5, and Q_6), we use a window of length 15 minutes and a slide of size 1 minute. Query Q_7 (i.e., the soccer query) uses a window of length 30 seconds and a slide of size 1 second. We stream events to the operator from the datasets stored in files. We first stream events at input event rates that 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 to have a safety bound, where they start dropping events when the event queuing latency is greater or equal to 80% of LB, i.e., the safety bound equals 200 milliseconds. We execute several runs for each experiment and show the mean value and standard deviation.

Several factors influence the performance of gSPICE, such as the event rate, event

Queries on synthetic data	
Q_1	**pattern seq**$(A; B; C)$ **where** $A.V_1 < B.V_1$ **and** $A.V_1 + B.V_1 < C.V_1$ **within** ws seconds
Q_2	**pattern seq**$(A; B; C) \vee$ **seq**$(D; E; F)$ **where** $(A.V_1 < B.V_1$ **and** $A.V_1 + B.V1 < C.V_1)$ **or** $(D.V_1 < E.V_1$ **and** $D.V_1 + E.V1 < F.V_1)$ **within** ws seconds
Q_3	**pattern seq**$(A; B+; C)$ **where** $A.V_1 + \sum_{i<j} B_i.V_1 < B_j.V_1$ **and** $A.V_1 + \sum B.V_1 < C.V_1$ **within** ws seconds
Stock queries	
Q_4	**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_5	**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_6	**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 query	
Q_7	**pattern seq**$(S;$ **any**$(3, D_1, D_2, .., D_m))$ **where** S *possesses ball* **and** *distance*$(S, D_i) \leq x$ *meters* , $i = 1..m$ and m is the number of players in a team **within** ws seconds

Table 6.4: Queries.



(a) Q_1: strict QoR

(b) Q_2: strict QoR

(c) Q_1: relaxed QoR

(d) Q_3: strict QoR

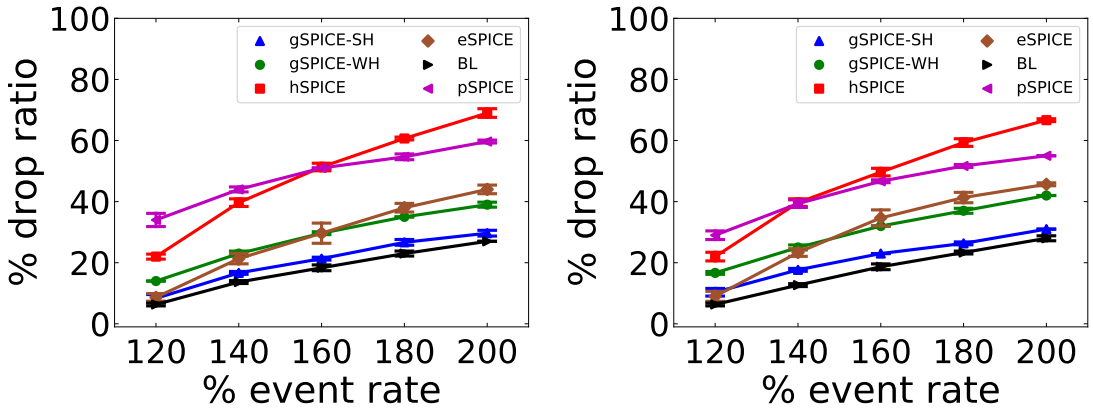
Figure 6.4: Synthetic: Impact of event rate on false negatives.

distribution, and the predecessor pane length L_ω. Therefore, next, we analyze the performance of gSPICE with these different factors.

6.3.2.1 Results on Synthetic Data

To evaluate the performance of gSPICE, we run experiments with queries Q_1, Q_2, and Q_3 with event rates 120%, 140%, 160%, 180%, and 200% of the operator throughput μ (i.e., the input event rate is higher than the operator throughput μ by 20%, 40%, 60%, 80%, and 100%). For queries Q_1 and Q_3, we use the dataset DS_1. While for query Q_2, we use the dataset DS_5. For all queries, we use a predecessor pane of length 10 events, i.e., $L_\omega = 10$.

Impact on False Negatives. Figure 6.4 shows the shedding impact with different event rates on the percentage of false negatives for all queries suing strict QoR. Moreover, in the figure, we show the percentage of false negatives using relaxed QoR for query Q_2. We observe similar results for Q_1 and Q_3 when using relaxed QoR, hence we do not show them. While Figure 6.5 depicts the ratio of dropped events or PMs (for pSPICE) with different event rates for queries Q_1 and Q_3. The drop ratio indicates the overhead

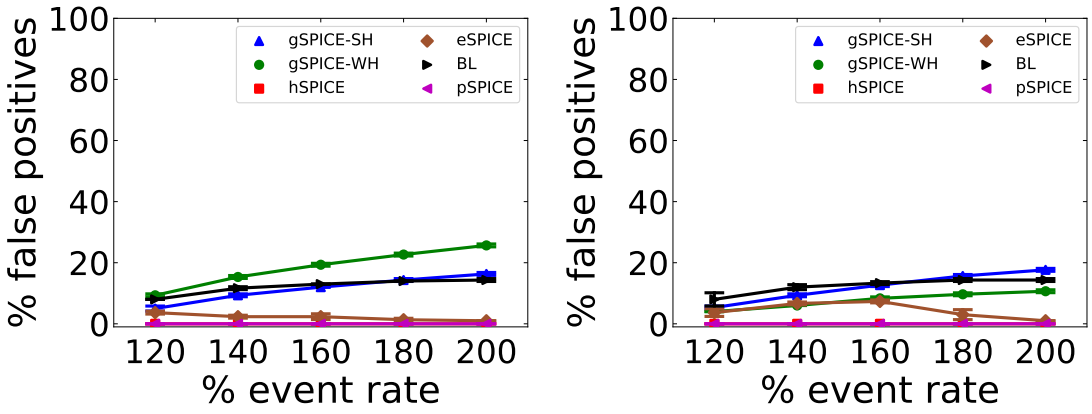


(a) Q_1 **(b)** Q_3

Figure 6.5: Synthetic: Impact of event rate on drop ratio.

of a load shedding strategy. We observed similar results, w.r.t. drop ratio, for Q_2, hence we do not show them. In both figures, the x-axis represents the event rate. The y-axis in Figure 6.4 represents the percentage of false negatives, while, in Figure 6.5, it represents the ratio of dropped events/PMs.

Increasing the event rate increases the overload on the operator, thus increasing the need to drop more events. Dropping more events might increase the percentage of false negatives. Figure 6.4a depicts results for $Q_1$when using strict QoR. In the figure, the impact of gSPICE-SH on false negatives is almost negligible irrespective of the used event rate. However, the percentage of false negatives caused by gSPICE-WH increases from 5% to 13% when increasing the event rate from 120% to 200%. The reason behind this is that gSPICE-WH has a higher overhead than gSPICE-SH, where, for an event e in the input event stream, gSPICE-WH must take the shedding decision for event e individually within each window to which event e belongs. This is also shown in Figure 6.5a where gSPICE-WH drops almost up to 1.75 times more events than gSPICE-SH. In Figure 6.4a, hSPICE performs, w.r.t. false negatives, very good where it has almost zero impact on the false negatives. The percentage of false negatives caused by eSPICE, BL, and pSPICE increases from 19% to 53%, 11% to 45%, and 8% to 32% when increasing the event rate from 120% to 200%, respectively. That shows that eSPICE suffers from a relatively high percentage of false negatives. The reason behind this is that eSPICE assumes that there exists a correlation between event types in the dataset. However, the event types in the used dataset (i.e., DS_1) do not have correlations. The results show that gSPICE-SH significantly outperforms, w.r.t. false negatives, gSPICE-WH, eSPICE, BL, and pSPICE for Q_1. That is because gSPICE-SH uses complex features, such as the type frequency and event attributes, that improve the prediction accuracy. The performance, w.r.t. false negatives, of hSPICE is comparable to the performance of gSPICE-SH. Although gSPICE-SH uses complex features, Figure 6.5a shows that gSPICE-SH has a relatively low drop ratio compared to other load shedding approaches,



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

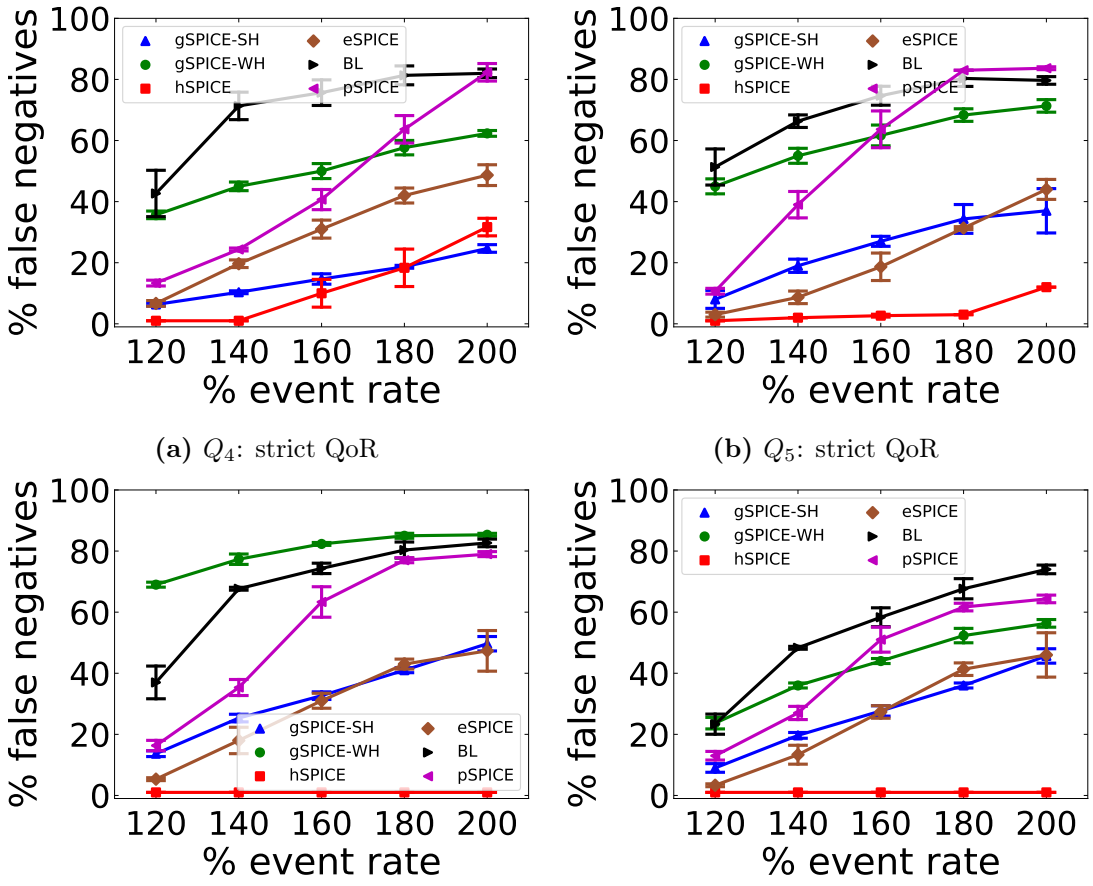
Figure 6.6: Synthetic: Impact of event rate on false positives.

where its drop ratio is comparable to the drop ratio of BL. That shows that gSPICE-SH is a lightweight load shedding approach.

The results for Q_2 using strict QoR are depicted in Figure 6.4b. The percentage of false negatives caused by gSPICE-SH only slightly increases when increasing the event rate. In the figure, hSPICE has almost no impact on the false negatives. While the percentage of false negatives caused by gSPICE-WH, eSPICE, BL, and pSPICE increases when increasing the event rate. Using relaxed QoR, the impact of gSPICE-SH, gSPICE-WH, and BL on the percentage of false negatives decreases compared to the case when using strict QoR, as depicted in Figure 6.4c. The performance of eSPICE only slightly improves when using relaxed QoR. Using relaxed QoR might improve the performance, w.r.t. QoR, of a load shedding approach since it does not require that the exact event instances match the defined pattern, as is the case in the strict QoR. Figure 6.4d shows results for query Q_3 when using strict QoR. The figure shows that gSPICE-SH, again, has a good performance where it results in almost zero false negatives. Similar to the results of Q_1, the results show that gSPICE-SH outperforms, w.r.t. false negatives, gSPICE-WH, eSPICE, BL, and pSPICE for Q_2 and Q_3.

Impact on False Positives. Shedding events might result in false positives when using strict QoR. However, when using relaxed QoR, shedding events might result in false positives only in the case of using the negation event operator (cf. Chapter 2, Section 2.2). As queries Q_1, Q_2, and Q_3 do not use the negation event operator, shedding events for these queries does not cause false positives when using relaxed QoR. Therefore, next, we show the impact of gSPICE on false positives only when using strict QoR. Figure 6.6 shows the shedding impact on the false positives for queries Q_1 and Q_2. In the figure, the x-axis represents the event rate, while the y-axis represents the percentage of false positives. We observe similar results for Q_3, hence we do not show them.

Figure 6.6a depicts results for Q_1 showing that the percentage of false positives caused by gSPICE-SH, gSPICE-WH, and BL increases when increasing the event rate. However,



(a) Q_4: strict QoR

(b) Q_5: strict QoR

(c) Q_6: strict QoR

(d) Q_6: relaxed QoR

Figure 6.7: Stock: Impact of event rate on false negatives.

the impact of hSPICE, eSPICE, and pSPICE on false positives in Q_1 is negligible, as shown in Figure 6.6a. hSPICE and eSPICE outperforms, w.r.t. false positives, gSPICE-SH since both hSPICE and eSPICE, in contrast to gSPICE-SH, consider the order of events in windows when predicting the event utilities, which has a considerable impact on the false positives. Considering event orders enables hSPICE and eSPICE to assign to event instances of the same event type different utilities depending on their probability to match the pattern, where event instances that are more likely to match the pattern are assigned higher utilities. Figure 6.6b shows that the results for Q_2 have similar behavior.

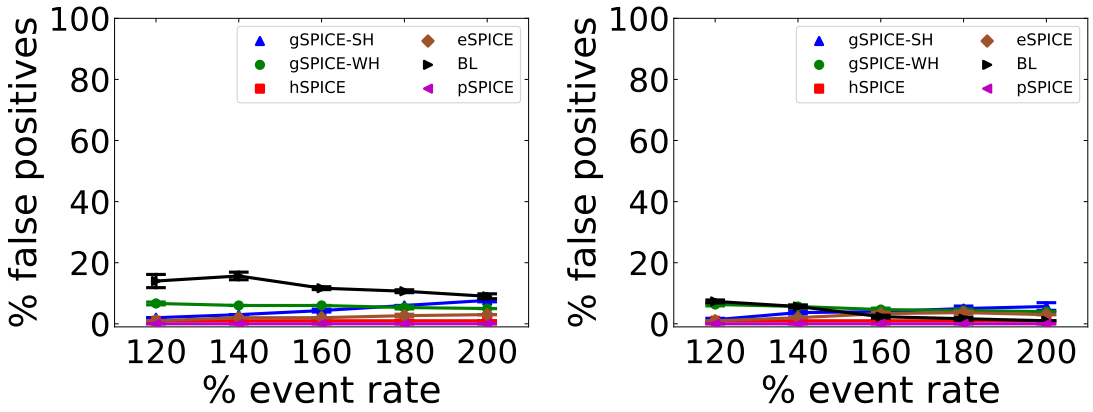
6.3.2.2 Stock Results

Now, we show the results obtained from evaluating gSPICE over the NYSE dataset. We run experiments with queries Q_4, Q_5, and Q_6, where gSPICE uses a predecessor pane of length 50 events, i.e., $L_\omega = 50$. Moreover, we stream the NYSE dataset to the operator using the following input event rates: 120%, 140%, 160%, 180%, and 200% of the operator throughput μ.

Impact on False Negatives. Figure 6.7 shows the impact of load shedding on false negatives for queries Q_4, Q_5, and Q_6 when using different event rates. The figure shows results for all queries when using strict QoR and the results for Q_6 when using relaxed QoR. We observe similar results for Q_4 and Q_5 when using relaxed QoR, hence we do not show them. In the figure, the x-axis represents the event rate, while the y-axis represents the percentage of false negatives.

Figure 6.7a shows results for Q_4 when using strict QoR. The percentage of false negatives caused by the load shedders increases when increasing the event rate. The percentage of false negatives caused by gSPICE-SH increases from 6% to 24% when the event rate increases from 120% to 200%. gSPICE-SH performs, w.r.t. the percentage of false negatives, better than all other load shedders when the event rate is higher than 180%. The percentage of false negatives caused by hSPICE is almost zero when the event rate is lower than 140%. After that, it increases to reach 32% when the event rate is 200%. For other load shedders, gSPICE-WH, BL, eSPICE, pSPICE, the percentage of false negatives increases from 36% to 62%, 43% to 82%, 7% to 49%, and 13% to 82% when the event rate increases from 120% to 200%, respectively. Again, the results show that using the type frequency and event attributes in gSPICE-SH improves the accuracy of predicted event utilities. However, the performance, w.r.t. false negatives, of gSPICE-SH and gSPICE-WH is worse than their performance when using synthetic data. That is because, in Q_4, gSPICE matches stock events that might have an increase or decrease in their quotes (i.e., attribute values). Hence in Q_4, the event attributes provide less useful information to predict the event utilities compared to event attributes in queries on synthetic data. The results show that gSPICE-SH performs, w.r.t. false negatives, better than gSPICE-WH, BL, eSPICE, and pSPICE by up to 6, 7.2, 2, and 3.4 times, respectively. Moreover, for high event rates, gSPICE-SH outperforms hSPICE, where gSPICE-SH has a better performance than hSPICE by up to 1.3 times.

The results for query Q_5 show similar behavior to Q_4, as depicted in Figure 6.7b. However, the figure shows that the performance, w.r.t. false negatives, of gSPICE-SH for Q_5 is worse than its performance for Q_4. The reason behind this is that in query Q_5 event types repeat. For two events e_1 and e_2 of the same type that repeats in Q_5 (i.e., $T_{e_1} = T_{e_2}$), even with the same type frequency F and event attributes $\mathbb{E}_e$ (i.e., $\mathbb{E}_{e_1} = \mathbb{E}_{e_2}$), the match probability of event e_1 and e_2 might be different depending on their position in query Q_5. However, for gSPICE-SH both events e_1 and e_2 get the same utility value since the following features are the same for both events e_1 and e_2: $T_{e_1} = T_{e_2}$, the same type frequency F, and $\mathbb{E}_{e_1} = \mathbb{E}_{e_2}$. That may impact the ability of gSPICE-SH to accurately predict the event utilities. Hence, it might influence its impact on QoR. Figure 6.7b shows that gSPICE-SH outperforms, w.r.t. false negatives, gSPICE-WH, BL, and pSPICE irrespective of the used event rate. However, gSPICE-SH outperforms eSPICE only with high input event rates. Moreover, the figure shows that hSPICE performs very well with the query Q_5, where it outperforms gSPICE,



(a) Q_4: strict QoR **(b)** Q_6: strict QoR

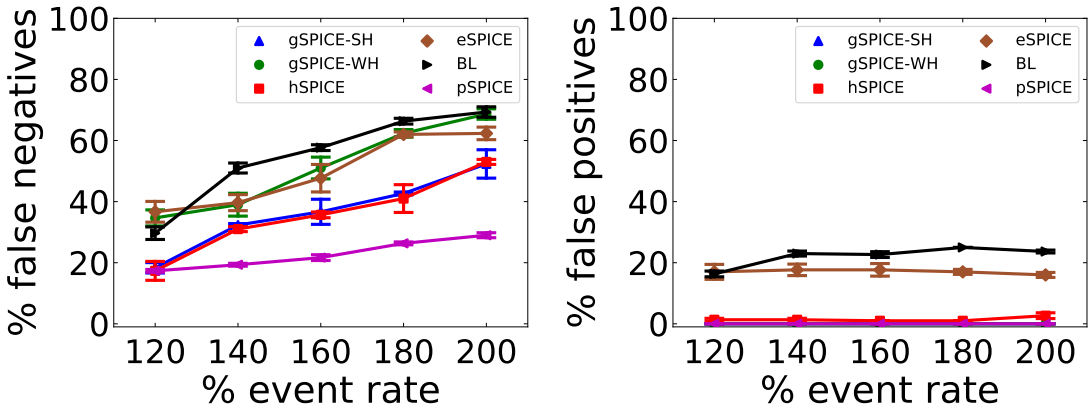
Figure 6.8: Stock: Impact of event rate on false positives.

irrespective of the used event rate.

Figures 6.7c and 6.7d show results for Q_6 using strict and relaxed QoR, respectively. Figure 6.7c shows that the percentage of false negatives for all load shedders, except hSPICE, increases when increasing the event rate. The performance of gSPICE-SH and gSPICE-WH with Q_6 is worse than their performance with Q_4 due to the following. Since Q_4 contains the negation event operator, gSPICE might assign to event types in Q_6 that are before the negated event type (i.e., C_5) higher utilities than the event types that are after the negated event type. That might negatively influence the ability of gSPICE to correctly drop events, thus, increasing its impact on QoR. Figure 6.7c shows that eSPICE outperforms, w.r.t. false negatives, gSPICE-SH with low input event rates. For an input event rate that is equal to or higher than 160%, the performance of gSPICE-SH is comparable to the performance of eSPICE. hSPICE has almost zero impact on the percentage of false negatives. The results show that gSPICE-SH has considerably better performance, w.r.t. false negatives, compared to gSPICE-WM, BL, and pSPICE. When using relaxed QoR, the percentage of false negatives caused by gSPICE-WH, BL, and pSPICE considerably decreases, as depicted in Figure 6.7d. In this case (i.e., using relaxed QoR), gSPICE-WH outperforms BL and, for high event rates, it outperforms pSPICE as well. The percentage of false negatives caused by gSPICE-SH and eSPICE only slightly decreases compared to the case when using strict QoR.

Impact on False Positives. Figure 6.8 shows the percentage of false positives for queries Q_4 and Q_6 using strict QoR. We observed similar results for Q_6, hence we do not show them. In the figure, the x-axis represents the event rate, while the y-axis represents the percentage of false positives.

The percentage of false positives caused by gSPICE-SH and eSPICE slightly increases when increasing the event rate, as depicted in Figure 6.8a. The percentage of false positives caused by gSPICE-WH slightly decreases when increasing the event rate. Moreover, the figure also shows that the percentage of false positives caused by BL



(a) Q_7: false negatives

(b) Q_7: false positives

Figure 6.9: Soccer: Impact of event rate on QoR.

slightly increases when the event rate increases from 120% to 140%. After that, the percentage of false positives decreases when the event rate increases. That is because gSPICE-WH and BL result in a high number of false negatives with high event rates. A high number of false negatives may imply that only a small number of complex events are detected that may result in a low percentage of false positives. The results for query Q_5 show similar behavior to the results of Q_4, as depicted in Figure 6.8b.

6.3.2.3 Soccer Results

Next, we analyze the performance of gSPICE on the RTLS dataset. Please note, since event types in the RTLS dataset occur periodically (cf. 6.3.1), the predecessor pane ω_e may not help predict the event utilities as all event types will have, on average, the same frequency in the type frequency F. We run experiments with query Q_7 using a pane of length 200 events, i.e., $L_\omega = 200$, and the following event rates: 120%, 140%, 160%, 180%, and 200% of the operator throughput μ. Figures 6.9a and 6.9b show the impact of load shedding on false negatives and positives for Q_7 when using strict QoR, respectively. We observe similar results for the impact of shedding on false negatives when using relaxed QoR, hence we do not show them. In both figures, the x-axis represents the event rate. The y-axis in Figure 6.9a represents the percentage of the false negatives, while in 6.9b, it represents the percentage of false positives.

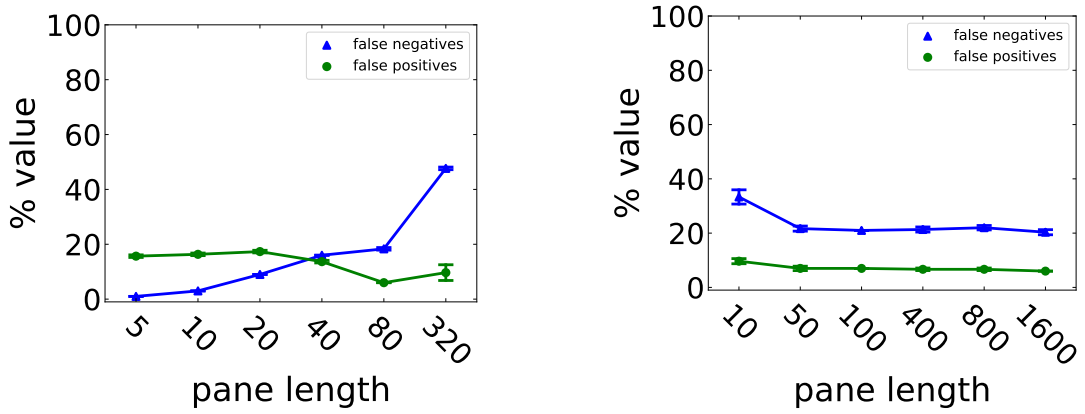
The percentage of false negatives caused by the load shedders increases when the increasing event rate (cf. Figure 6.9a). Q_7 contains the *any* event operator, where any event type (i.e., any defender from the opposite team) may match the pattern. Hence, the event utilities are more spread out, and it is hard to accurately predict the utilities for different event types. However, the figure shows that gSPICE-SH still outperforms, w.r.t. false negatives, gSPICE-WH, BL, and eSPICE, irrespective of the used event rate. Moreover, the performance of gSPICE-SH is similar to the performance of hSPICE. However, the figure shows that pSPICE outperforms, w.r.t. false negatives, gSPICE-SH

with high input event rates. Figure 6.9b shows that gSPICE-SH, gSPICE-WH, hSPICE, and pSPICE result in almost zero false positives. The percentage of false positives caused by eSPICE slightly decreases when the event rate increases, while the percentage of false positives caused by BL slightly increases when increasing the event rate. The results show that gSPICE-SH has a relatively good performance, w.r.t. QoR, even when the predecessor pane ω_e is not very useful for predicting the event utilities. That implies that the other two features (i.e., the event type and event attributes) used to predict the event utilities in gSPICE-SH are important features.

6.3.2.4 Impact of Predecessor Pane Length on QoR

The pane length may considerably impact the utility prediction. Hence, it may influence the impact of gSPICE on QoR. For an event e, the pane length defines the number of past incoming events that might have an impact on the importance of event e. If the length of the predecessor pane is too small, gSPICE may not be able to capture the events that influence the utility of the event e. On the other hand, if the length of the predecessor pane is too large, the predecessor pane ω_e might contain many unrelated events (i.e., noisy data) that might hinder accurately predicting the event utilities. Moreover, a large predecessor pane might increase the overhead of gSPICE, thus negatively impacting QoR. To evaluate the impact of pane length on the performance, w.r.t. QoR, of gSPICE, we run experiments with Q_2 and Q_4, where we run Q_2 over dataset DS_5. For Q_2, we use a pane of the following lengths: 5, 10, 20, 40, 80, 320. While for Q_4, we use a pane of the following lengths: 10, 50, 100, 400, 800, 1600. Moreover, for both queries, we use a fixed event rate of 180% of the operator throughput μ. Figure 6.10 depicts results of gSPICE-SH for both queries using strict QoR. The results for gSPICE-WH show similar behavior, hence we do not show them. In the figure, the x-axis represents the predecessor pane length, while the y-axis represents the percentage of false negatives and positives.

Figure 6.10a depicts the results for Q_2, where it shows that increasing the pane length results in increasing the percentage of false negatives. Moreover, the figure shows that the impact of gSPICE-SH on the false positives decreases when slightly increasing the pane length. However, gSPICE-SH results in more false positives with a large pane length. For Q_4, gSPICE-SH has a high impact on the false negatives and positives with small pane lengths, as depicted in Figure 6.10b. However, increasing the pane length thereafter barely changes the incurred false negatives and positives. As a result, we may conclude that using the right pane length may influence the impact of gSPICE on QoR, where the right pane length depends on the used query and data. Hence, to select the best pane length that improves the performance, w.r.t. QoR, of gSPICE, we need to profile the performance of gSPICE with different possible pane lengths.



(a) Q_2 **(b)** Q_4

Figure 6.10: Impact of the predecessor pane length on QoR.

6.3.2.5 Impact of Event Distribution on QoR

The event distribution may considerably impact the performance, w.r.t. QoR, of gSPICE due to the following. The predecessor pane ω_e, which represents an important feature to predict the event utility loses its importance when all event types occur with the same frequency in a dataset. That implies, the type frequency F in the predecessor pane ω_e will almost always be the same. Hence, the type frequency will not help predict the event utilities. To evaluate this, we run experiments with all queries on the synthetic data. For all queries, we use a predecessor pane of length 10 events, i.e., $L_\omega = 10$, and a fixed event rate of 140% of the operator throughput μ. As mentioned in Section 6.2.2, gSPICE may use machine learning models to estimate the event utilities. Therefore, we also show the performance of gSPICE when using a decision tree or a random forest to predict the event utilities. We refer to gSPICE when events are dropped on the stream level as gSPICE-ST and gSPICE-SF when using a decision tree and a random forest, respectively. Similarly, We refer to gSPICE when events are dropped from windows as gSPICE-WT and gSPICE-WF when a decision tree and a random forest are used to predict the event utilities, respectively. In our experiments, the random forest consists of ten trees. Figure 6.11 depicts the impact of gSPICE on QoR for query Q_2, and Figure 6.12 shows the corresponding drop ratio. We observe similar behavior for other queries, hence we do not show them.

Figures 6.11a and 6.11b depict the shedding impact on false negatives and positives using strict QoR, respectively. We observe similar behavior for false negatives when using relaxed QoR, hence we do not show them. In both figures, the x-axis represents the used datasets. The y-axis in Figure 6.11a represents the percentage of false negatives, while it represents the percentage of false positives in Figure 6.11b. The x-axis in Figure 6.12 represents the used dataset, while the y-axis represents the drop ratio. The results show that for all variants of gSPICE, the percentage of false negatives is the lowest when using the dataset DS_5 and the highest when using the dataset DS_8 (cf. Figure 6.11a). In dataset DS_5, there exists a high difference between the frequency of event

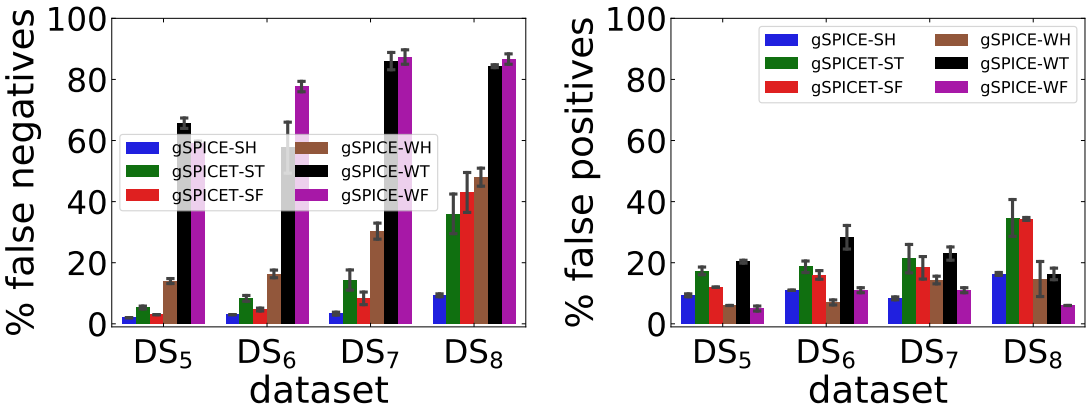
types. For example, events of type A are expected to form 40.7% of events in DS_5, while events of type C represent only 2.5% (cf. Table 6.3). This large difference between the amount of each event type enables the predecessor pane ω_e (i.e., the type frequency F) to contain more useful information that helps predict event utilities. While in DS_8, all event types occur at the same frequency on average. Hence, for dataset DS_8, the predecessor pane ω_e is not a useful indicator of the importance of event e. Figure 6.11a also shows that using datasets DS_6 and DS_7, the percentage of false negatives caused by all load shedders, is higher compared to the case when using dataset DS_5.

As Figure 6.11a shows, the performance, w.r.t. false negatives, of gSPICE-SF is better than the performance of gSPICE-ST irrespective of the used dataset. Moreover, the performance of gSPICE-SF is comparable to the performance of gSPICE-SH with datasets DS_5 and DS_6. That means that in the case of limited available memory, gSPICE-SF might be used as a replacement of gSPICE-SH with only a slight impact on QoR for these distributions. The performance of gSPICE-SF with DS_7 and DS_8 is worse than the performance of gSPICE-SH, especially with DS_8. Moreover, gSPICE-SH outperforms, w.r.t. false negatives, gSPICE-ST, irrespective of the used dataset. That is because gSPICE-SF and gSPICE-ST result in a high drop ratio compared to gSPICE-SH (cf. Figure 6.12). A high drop ratio implies that more events are dropped, hence negatively impacting QoR. In Figure 6.11a, the percentage of false negatives caused by gSPICE-WT and gSPICE-WF is very higher. That is because the overhead of these two load shedders is very high, where, as depicted in Figure 6.12, these two shedders result in a high drop ratio. As a result, using a decision tree or a random forest to drop events on the window granularity is not recommend due to their very high overhead.

The percentage of false positives may depend on the percentage of false negatives. If a load shedder results in a very high number of false negatives, this may imply that only a few complex events are detected. Having a low number of detected complex events may result in having a low number of false positives. We can observe this in Figure 6.11b. In the figure, the percentage of false positives caused by gSPICE-WT and gSPICE-WF first increases when using the dataset DS_6 compared to the case when using the dataset DS_5. Then, the percentage of false positives caused by these two shedders decreases when using the datasets DS_7 and DS_8 since these two shedders result in a high percentage of false negatives with the datasets DS_7 and DS_8 (cf. 6.11a). Figure 6.11b also shows that the percentage of false positives caused by gSPICE-SH and gSPICE-WH only slight changes when using different datasets. While the percentage of false positives caused gSPICE-ST and gSPICE-SF considerably increases when using the dataset DS_8 compared to the case when using the dataset DS_5.

6.3.2.6 Maintaining Latency Bound

gSPICE performs load shedding to maintain a given latency bound (LB). Figure 6.13 shows the ability of gSPICE to maintain the given latency bound, where it depicts



(a) Q_2: false negatives

(b) Q_2: false positives

Figure 6.11: Impact of event distribution on QoR.

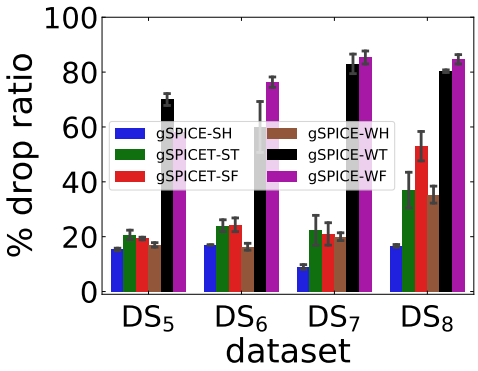
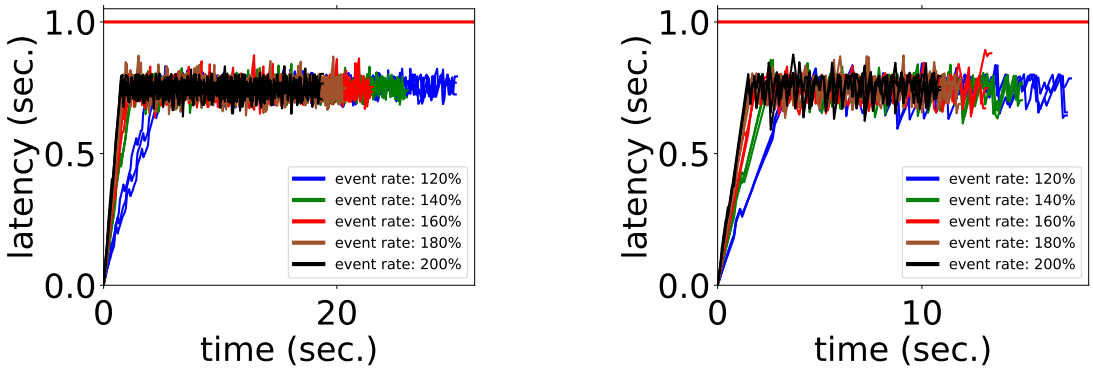


Figure 6.12: Impact of event distribution on drop ratio.

results for Q_1 and Q_4. We observe similar results for other queries, hence we do not show them. For all queries, we use the same setting as explained in Sections 6.3.2.1, 6.3.2.2, and 6.3.2.3. The figure shows that gSPICE always maintains the given latency bound, irrespective of the event rate. The induced event latency stays around 800 milliseconds (i.e., 80% of LB that represents a safety bound, as we mentioned above). That shows that gSPICE can successfully maintain a given latency bound.

6.3.2.7 Discussion

Through extensive evaluations with several datasets and a set of representative CEP queries, gSPICE shows that it has a good performance, w.r.t. QoR. For the majority of queries and datasets, gSPICE outperforms pSPICE and state-of-the-art black-box load shedding approaches (i.e., eSPICE and BL). gSPICE performs especially well when the event types do not follow a uniform distribution and when using the sequence event operator. However, hSPICE performs better than gSPICE for the majority of cases. Moreover, with low input event rates, eSPICE may outperform, w.r.t. QoR, gSPICE, e.g., when using the negation event operator. Furthermore, pSPICE outperforms gSPICE with



(a) Q_1 **(b)** Q_4

Figure 6.13: Maintaining latency bound.

high input event rates when using the *any* event operator. We also show that gSPICE-SH always outperforms gSPICE-WH since gSPICE-WH imposes a higher overhead. Moreover, the results show that using the right predecessor pane length may improve the performance of gSPICE considerably. Further, gSPICE is a lightweight load shedding approach as the overhead of performing load shedding in gSPICE-SH is low. We also show that to reduce the required memory, gSPICE may use well-known machine learning models, e.g., random forests, with a slight adverse impact on QoR, as these models impose a higher overhead.

6.4 Conclusion

In this chapter, we proposed an efficient black-box load shedding approach (called gSPICE) that drops events in overload cases to maintain a given latency bound. gSPICE drops events on the window and stream granularities. gSPICE uses a probabilistic model to predict the event utilities, where the event utilities are predicted depending on the following features: 1) event type, 2) event context, 3) event attributes, 4) event position within the window (when dropping events on the window granularity). To store event utilities, both gSPICE-SH and gSPICE-WH use the Zobrist hashing. Moreover, if the utility table is very large, to minimize the needed memory for storing the utilities, gSPICE may use a machine learning model (e.g., decision trees or random forests) to estimate the event utilities. Through extensive evaluations on several representative CEP queries and several synthetic and real-world datasets, we show that, for the majority of cases, gSPICE outperforms, w.r.t. QoR, pSPICE and state-of-the-art black-box load shedding strategies. Moreover, the results show that gSPICE-SH has better performance, w.r.t. QoR, than gSPICE-WH as gSPICE-WH imposes a higher overhead. Additionally, we show that gSPICE always maintains the given latency bound regardless of the incoming input event rate.

Chapter 7

Related Work

In this chapter, we discuss existing related works that are relevant to our research. First, we present the work performed in the area of complex event processing and the proposed techniques to handle a high volume of input event streams. Then, we discuss the available work on load shedding and data sampling in the stream processing and complex event processing domains. After that, we present work performed in the area of approximate event processing. Finally, we discuss the past work on handling uncertainty in the input event streams.

7.1 Complex Event Processing

Complex event processing (CEP) has emerged as a powerful paradigm to process event streams on the fly [Luc01; DGP07; WDR06; GJS92; CM94; CM12]. To formally define CEP patterns, researchers proposed several event definition languages [CM94; CM10; WDR06]. These event languages specify the rules to define CEP patterns, where they introduce multiple event operators that help to define CEP patterns. Examples of event operators are sequence, negation, any, Kleene closure, disjunction, and conjunction operators. In CEP, multiple event instances of the same event type might occur in the input event stream. To precisely match events in the input event streams, [CM94; CM10; ZU99] introduce selection and consumption policies. These policies accurately define event instances that must match a pattern and whether an event instance can be used in multiple complex events. Moreover, since the input event stream is continuous and infinite, it is common in CEP to partition the input event stream into windows of events, where a window represents a temporal constraint [Bal+13; WDR06; CM10; DP18]. This model is called *window-based* CEP, where events within a window are correlated together to detect complex events. The input event stream in CEP has a high volume and usually needs to be processed in near real-time [Quo+17; CF+13].

To process the incoming input events within a given latency bound, researchers have proposed several techniques such as parallelism, optimizations, and pattern sharing.

In [May+17; Bal+13; ZR10; Sha+03; MKR15; Apa; BTÖ13], the authors propose to distribute the CEP operator graph on multiple compute nodes and to parallelize each operator on one (scale-up) or more nodes (scale-out). In CEP, a powerful parallelization technique is data-parallel CEP. Data-parallel CEP is mainly divided into two categories, namely, window-based parallelization and key-based parallelization [RM19]. In [Bal+13; May+17; MKR15; Apa], the authors propose to process windows of events in parallel using multiple threads or compute nodes to increase the event processing throughput. In key-based parallelization [CF+13; BTÖ13; Apa], the incoming event stream is partitioned into groups depending on keys, where groups are then processed on multiple threads or compute nodes. The key-based parallelization is limited to the number of different keys. To efficiently process patterns, in [WDR06; WTA10; Hir+14; Hir12], the authors propose several optimizations, e.g., intra- and inter-operator optimizations [WDR06], or using special hardware (FPGA) to speed up the event processing [WTA10]. Another way to improve the operator throughput is by sharing the pattern matching between several patterns as proposed in [RLR16; SMMP09]. The authors propose algorithms to find the best sharing between different patterns in an operator.

7.2 Load Shedding

The above mentioned techniques (i.e., parallelization, optimizations, and pattern sharing) used to handle a high incoming input event rate may not always be possible or sufficient. In that case, load shedding may be used to enable an operator to process a high input event rate. Load shedding has been researched in various domains such as computer networking [Jac88; Jai90], multimedia streaming [Wal+99; BBS98], stream processing, and complex event processing. In this section, we focus on the work performed in the stream processing and CEP domains.

Load shedding in stream processing. Load shedding has been extensively studied in the stream processing domain [Car+02; Tat+03; OJW03; AN04; RH05; TZ06; TcZ07; CK09; WRM10; Mot+03; KLC18]. The idea is to drop tuples in a way that reduces the system load but still provides the maximum possible quality of results (QoR). Hence, the crucial question here is which tuples to drop so the quality of results is not impacted drastically. In [Car+02; Tat+03; TcZ07; KLC18], the authors assumed that the tuples have different utilities/importance and impact on QoR, where the utility of tuples depends on their content. In case of overload, tuples with low utility values are dropped. In [Car+02; Tat+03; TcZ07], the authors assume that the mapping between the utility and tuple's content is given, for example, by an application expert, while, in [KLC18], they learn this mapping online depending on the used query. Similarly, in [OJW03; GWY08; Mot+03], the authors propose filters that drop low important tuples. The work in [RBQ16] assigns utilities to tuples depending on their processing times. The higher is the processing time of a tuple, the lower is its utility. In overload cases, low utility tuples are dropped.

While in [CK09], the authors propose to assign utilities to tuples depending on the tuple frequency in the input event stream. The higher is the tuple frequency in the input event stream, the higher is its utility.

In [Quo+17], the authors assume that all tuples in the input event streams have the same utilities and processing latency. They fairly select tuples for dropping from different input streams by combining two techniques, namely, stratified sampling and reservoir sampling. The works in [Ged+05; Ged+07; SW04; DGR03] propose load shedding approaches for join operators where the goal is to increase the number of output tuples. Moreover, the authors in [TBL08] propose to use stratified sampling and reservoir sampling to perform an approximate join. In all the above works, the utilities of tuples are either computed using simple dependencies between tuples (e.g., in join operators) or they are computed for each tuple individually without considering the dependency between tuples at all. However, in the CEP domain, patterns are more complex than a simple binary join, where a pattern can be viewed as multi-relational non-equi-joins with temporal constraints [HBN14]. Moreover, events in a CEP pattern have interdependency with each other that we must take into consideration when assigning utilities to events. Additionally, the order of events in patterns and in the input event streams is important in CEP (e.g., in the sequence and negation event operators) that is not considered in the above works.

Load shedding in CEP. There exist only a few works on load shedding in the CEP domain. In [HBN14], the authors propose an event shedding approach for CEP systems. They formulate the load shedding problem in CEP as a set of different optimization problems, where they consider a multi-pattern operator. To assign utilities to events, the approach depends only on the event types, where the authors consider only the repetition of events in the input event stream and in patterns. However, they do not consider the order of events in both the input event stream and in patterns that is important in CEP, e.g., in the sequence and negation event operators. In [ZVHW20], the authors propose a load shedding approach to drop PMs and events. They assign utilities to PMs in a similar way to pSPICE (cf. Chapter 3), i.e., depending on the completion probability of PMs and their estimated processing cost. When load shedding is triggered, their 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 $\mathbb{E}_D$). 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 $e \in \mathbb{E}_D$, drop e– it drops events on the stream granularity. 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 a low utility may also contain highly important events. That 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. For example, in pattern $q = seq(A; B; C)$, events of type A may have higher probabilities to be a part of PMs than events of type B. As a result, events of type A may have higher probabilities to be part of the event shedding set $\mathbb{E}_D$. Moreover, this load shedding approach uses event attributes to check if an event belongs to the event shedding set. Using events with their attributes in the event shedding set might considerably increase the load shedding overhead. The load shedding overhead in our proposed approaches eSPICE, pSPICE, and hSPICE, on the other hand, is independent of the event attributes. Moreover, in gSPICE, we limit our scope to numerical event attributes and use bin sizes to reduce the load shedding overhead. Furthermore, the load shedding approach in [ZVHW20] seems to only support skip-till-any-match semantic [Agr+08] that is equivalent to the *each* selection policy and *zero* consumption policy. That represents a small set of the known pattern semantics in CEP [CM94; CM10; ZU99; WDR06]. Moreover, this approach does not support the negation operator. In contrast, our proposed load shedding approaches, in this thesis, support all commonly used event operators and the selection and consumption policies.

7.3 Approximate Event Processing

Approximate event processing and load shedding share some commonality where both techniques aim to handle a high volume of event streams in the case of limited resources. To fulfill defined constraints (e.g., latency bound, monetary cost, etc.), both techniques may result in inaccurate output results. In [Gil+01; MM02; Cor+04; GFS10], the authors propose approximate processing approaches that can minimize the needed space to store tuples and increase the throughput by means of summarization while at the same time reducing the inaccuracy in the query results. To minimize the space and time needed to detect duplicates in the event stream, the authors in [MAEA05] propose a probabilistic approach based on the Bloom filter [Blo70]. Similarly, the authors in [DNB13] propose an approximate approach that uses hashing to efficiently detect duplication in the input event stream.

The work in [LG15; LG16] presents approximate processing techniques for the CEP domain. In [LG15], the authors propose an approximation approach that allows matching events not occurring in the exact order, as defined by patterns. Their focus is on interleaving patterns where a pattern consists of many parallel branches. The authors in [LG16] propose an approximation 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.

7.4 Uncertainty in Event Processing

The uncertain event processing deals with the problem of missing or imprecise events. The input event stream might suffer from missing events or events with imprecise data due to several reasons, e.g., summarizing events to decrease their size, inaccurate event sources, and the unreliable transmission of events [LM04; Cug+15; AGS16; Man+18; Mor+19; KKL10; ZDI10; Let+10]. Hence, the uncertain event processing area has close relevance to the load shedding area. The queries in both areas should process imprecise input event streams, which might negatively impact the quality of the output results. In [LM04], the authors propose a solution for the problem of finding a set of events $\mathbb{A}$ that match defined predicates where events might be imprecise. They develop an approach that maintains a defined quality of results by probabilistically finding events that belong to the set $\mathbb{A}$. The work in [ZDI10] presents formal semantics of pattern matching under the temporal uncertainty model to tackle the problem of pattern matching over events with imprecise occurrence times.

The authors in [Cug+15] present a model called CEP2U to handle uncertainty in CEP. They assume that the uncertainty might be caused by either missing events or imprecise events. They assign probabilities to events indicating the confidence degree of the event occurrences. Moreover, to handle imprecise event attributes, they model the error in an event attribute using a known probability distribution function. Similarly, the authors in [Mor+19] develop a library that can be integrated with the currently available CEP frameworks to handle missing events and imprecise event attributes using probabilities. Similarly, in [Man+18], the authors design a framework on top of the Hadoop MapReduce to estimate the uncertainty ranges of the output. The uncertainty in the output is originated from processing imprecise events or performing approximate query processing. Instead of providing exact values as output, they produce outputs with error bounds (with a specific confidence level). This way, they could trade-off the result precision with the incurred latency, consumed energy, etc.

Chapter 8

Summary and Future Work

In this chapter, we summarize the main contributions of this thesis and provide an outlook on the possible feature work in the area of load shedding in CEP.

8.1 Summary

In this work, we proposed four load shedding approaches that enable CEP operators to maintain a given latency bound in overload cases. Our proposed load shedding approaches cover a wide range of load shedding classes in the CEP domain. Moreover, we presented two ways to measure the quality of results in CEP. In the following, we summarize our contributions.

- We defined two ways to measure the quality of results (QoR), namely the strict and relaxed quality of results. The strict QoR follows the strict semantics of event matching in CEP systems using selection and consumption policies. The strict QoR requires that exact event instances are used in detecting complex events. The relaxed QoR, on the other hand, focuses only on whether a complex event is detected or not without considering the exact event instances. Deciding which QoR measurement to use depends mainly on the application.

- A white-box load shedding approach (called pSPICE) to drop partial matches (PMs) from the internal state of a CEP operator. To minimize the shedding impact of pSPICE on QoR, we assign utilities to PMs and drop those PMs that have the lowest utilities. pSPICE uses the Markov chain and Markov reward process to predict the PM utilities. The utility of a PM depends on two features: the PM state and the number of remaining events in a window. pSPICE does not result in false positives when using relaxed QoR. Moreover, as we showed in our evaluations (cf. Chapters 4, 5, and 6), pSPICE results in a low percentage of false positives when using strict QoR. Hence, pSPICE may be a good choice in applications that cannot tolerate many false positives. Moreover, as our evaluations show when

using the *any* operator, the performance, w.r.t. QoR, of pSPICE is very good compared to other shedders. In the majority of cases, pSPICE outperforms the other load shedding approaches when using the *any* operator. We also developed an algorithm that depends on the current number of PMs in a CEP operator to decide when and how many PMs to drop from the operator. pSPICE is a lightweight load shedding approach, where it gets the utility of a PM in $Q(1)$ time complexity. Moreover, it performs the shedding of PMs with a very low overhead where, as we showed in Section 3.3, its overhead is comparable to the overhead of a random PM dropper.

- eSPICE, a black-box load shedding approach that, in overload cases, drops events from windows in the input queue of a CEP operator. To reduce the drop impact on QoR, eSPICE assigns utilities to events, where it drops events with the lowest utilities. In eSPICE, the event utility in a window depends on the following two features: the event type and the event position within the window. To predict the utility of events within windows, eSPICE uses a probabilistic model that depends on those two features. Our evaluations (cf. Section 4.3) show that for the majority of queries– especially for the *sequence* operators– eSPICE performs well. However, eSPICE assumes that there is a correlation between event types. If the evaluation dataset does not contain a correlation between event types, the performance of eSPICE might degrade (cf. Section 6.3). Moreover, we developed an approach that decides when to drop events and how many events to drop to maintain a given latency bound. Furthermore, eSPICE computes a utility threshold from the predicted utilities and the distribution of incoming input events. The utility threshold helps eSPICE to perform load shedding in a lightweight way. eSPICE gets the utility of events and performs load shedding in $Q(1)$ time complexity. Hence, its load shedding overhead is very low– this is also shown in Section 4.3.

- A white-box load shedding approach (called hSPICE) that combines the advantages of both pSPICE and eSPICE. hSPICE drops events either from windows or from PMs, i.e., it performs shedding on two different granularity levels: the window (i.e., hSPICEW) and the PM granularities (i.e., hSPICEPM). In hSPICE, we consider that an event has different importance for different PMs. Therefore, hSPICE assigns a utility to an event for each PM in a window. To learn about the utility of events for a PM, hSPICE uses the following three features: 1) event type, 2) event position within the window, and 3) the current state of the PM. hSPICE uses a probabilistic model to predict the event utilities depending on these three features. Our evaluations (cf. Sections 5.3 and 6.3) show that, for the majority of queries, hSPICE outperforms, w.r.t. QoR, pSPICE, eSPICE, gSPICE, and the other state-of-the-art load shedding approaches. pSPICE outperforms hSPICE only in the case of the *any* event operator. Although hSPICEPM gets the utility of

an event for a PM and performs the load shedding in $Q(1)$ time complexity, its load shedding overhead is high, as shown in Section 5.3, especially in the case of low event processing time. Moreover, using a very large window size may increase the overhead of hSPICEPM even further, hence increases its negative impact on QoR. However, in this case, hSPICEW might be used to reduce the negative impact of shedding on QoR.

- A black-box load shedding approach (called gSPICE) that drops events either from windows or from the input event stream of an operator, i.e., it sheds events on the window and stream granularities. gSPICE uses a probabilistic model that depends on the following features to predict the utility of an event: 1) event type, 2) type frequency in the predecessor pane, 3) event content/attributes, and 4) event position within the window (when dropping events on the window granularity). gSPICE uses complex features such as the type frequency and event content to improve the utility prediction accuracy. Moreover, gSPICE uses the Zobrist hashing to efficiently store the event utilities and perform load shedding with low overhead. Furthermore, if the utility table is very large, to minimize the needed memory for storing the utilities, gSPICE may use a machine learning model (e.g., decision trees or random forests) to estimate event utilities. Our evaluations (cf. Section 6.3) show that gSPICE, for the majority of queries, outperforms, w.r.t. QoR, pSPICE and state-of-the-art black-box load shedding approaches. The performance of gSPICE is especially good when events in an evaluation dataset do not follow a uniform distribution. Additionally, the evaluations show that gSPICE performs shedding with relatively low overhead (cf. Section 6.3)

To conclude, in this thesis, we proposed four load shedding approaches that cover a wide range of load shedding classes in CEP, where we proposed two black-box and two white-box shedding approaches. Our approaches perform shedding by dropping PMs or events. Moreover, dropping events is performed on three different granularity levels: the stream, the window, and the PM granularities. Additionally, our developed shedding approaches use several features to predict the utility of events and PMs. The performance of our proposed load shedding approaches depends on the following: the used application, the distribution of events in the input event stream, and the used queries.

8.2 Future Work

In this section, we shed light on the possible future research directions to extend the work presented in this thesis.

To minimize the shedding impact on QoR, we may profile our proposed load shedding approaches and choose the shedding approach that results in the lowest negative impact

on QoR. However, as the event distribution may influence the performance of our proposed shedding approaches, there is a need to re-profile the performance of the proposed shedding approaches when the event distribution changes. Therefore, an interesting future extension to our work is to develop a mechanism that adaptively selects the best shedding approach that results in the lowest negative impact on QoR. The developed mechanism may monitor the event distribution. If the event distribution changes by a certain threshold, it re-profiles the operator with the proposed load shedders and chooses the shedding approach that has the lowest adverse impact on QoR.

In our current work, we focused on minimizing the degradation in QoR while maintaining a given latency bound, in the case of limited resources. However, load shedding might be used to achieve other objectives and to fulfill other constraints. Other interesting objectives might be to reduce the induced monetary cost, increase the throughput, reduce the power consumption, etc. While the constraints might be to maintain a certain QoR, monetary budget, etc. Hence, a possible extension to our work is to study the load shedding problem with other objectives and constraints.

Finally, in this thesis, we focused on performing load shedding in each operator in the CEP operator graph without considering the impact between the operators in the operator graph, i.e., we focused on optimizing the shedding locally in each operator without considering a globally optimized solution. However, aiming for a globally optimized load shedding solution requires that we consider the following questions that directly influence the shedding impact on the quality of results (QoR). 1) Where to shed load in the operator graph and how much load to shed from each operator? The answer to this question depends on the criticality of the output of each operator. The higher is the importance of the output of an operator, the less load should be shed from the operator. 2) How to distribute the available end-to-end latency between different operators in the operator graph?. This question is directly related to question 1, where the assigned latency to an operator may depend on the amount of load that should be shed from the operator. 3) How to achieve an optimal allocation of available resources and where to place operators in the case of distributed computing resources? This question also is related to questions 1 and 2, where the allocation and placement of resources may depend on the available latency bound for each operator. There exists extensive work in the literature on the allocation of resources and placement of the operator graph [YBT05; CLN12; EZ+13; KSP14]. That might be of great help to develop an optimal allocation and placement strategy of the operator graph that minimizes the negative shedding impact on QoR. 4) How to consider the dependency between operators in the operator graph to perform an optimal shedding? Performing load shedding in each operator locally without considering the dependency between operators in the operator graph might result in a sub-optimal solution. To clarify this, let us introduce an example of an operator graph that is depicted in Figure 8.1, where the operator graph consists of two event producers (pr_1 and pr_2), four operators (op_1, op_2, op_3, and op_4), and one

event consumer (cr_1).

In Figure 8.1, operator op_3 matches pattern $q_1 = any(3, A, A, B)$, where it receives events of type A and B from the upstream operators op_1 and op_2, respectively. The operator op_4 matches pattern $q_2 = any(2, C, D)$, where it receives events of type C and D from the upstream operators op_1 and op_2, respectively. In this example, the *any* event operator represents the occurrence of all event types, ignoring the order of their occurrences [CM94]. Moreover, Q_3 and Q_4 represent complex events detected in operators op_3 and op_4, respectively. Assume that the rate of events produced by operators op_1 and op_2 is 100 events per second for each event type (i.e., A, B, C, and D). Additionally, assume that the strict QoR is used and operators op_3 and op_4 match events chronologically, i.e., using the *first* selection policy and *consumed* consumption policy for each event type [CM94]– an event instance of any type might be a part of only one complex event.

In this example, operators op_3 and op_4 might detect 50 complex events (Q_3) per second and 100 complex events (Q_4) per second, respectively. Therefore, in operator op_3 there exist 50 events of type B per second not used in any complex events, hence dropping these events has no impact on QoR. Now, assume that there is an overload on the operator op_2, and there is a need to shed load. Shedding load in operator op_2 might hinder the detection of events of types B and D. Since operator op_3 needs only 50 events of type B per second, operator op_2 might first shed load that results in producing those 50 events of type B per second without negatively influencing QoR. However, operator op_2 locally has no knowledge about the number of B events needed in operator op_3. Hence, it might shed load that hinders the detection of events of type D that will adversely impact the final QoR. Assigning weights in operator op_2 manually by a domain expert to give more importance to a pattern that generates events of type D is only possible if the stream is steady, i.e., the distribution of events does not change. However, in real-world applications, that mostly does not hold, where the event distribution changes over time. Hence, there is a need to continuously adapt the pattern weights to control the drop amount of each event type in each operator. That needs a global solution that monitors the entire operator graph and figures out which events/patterns are more important and have more impact on QoR.

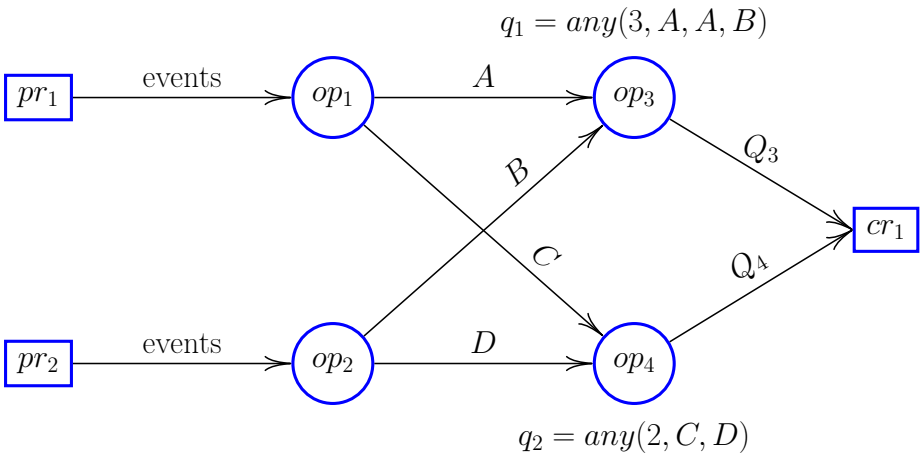


Figure 8.1: Dependencies between operators in the operator graph.

Bibliography

[ABO20] Matteo Aquilina, Eric B Budish, and Peter O'Neill. "Quantifying the high-frequency trading "arms race": A simple new methodology and estimates." In: *Chicago Booth Research Paper* 20-16 (2020).

[AC04] Raman Adaikkalavan and Sharma Chakravarthy. "Formalization and Detection of Events over a Sliding Window in Active Databases Using Interval-Based Semantics." In: *ADBIS*. 2004.

[AC06] Raman Adaikkalavan and Sharma Chakravarthy. "SnoopIB: Interval-based event specification and detection for active databases." In: *Data & Knowledge Engineering* 59.1 (2006), pp. 139–165.

[Agr+08] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. "Efficient Pattern Matching over Event Streams." In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. Vancouver, Canada: Association for Computing Machinery, 2008, 147–160.

[AGS16] V. Akila, V. Govindasamy, and S. Sandosh. "Complex event processing over uncertain events: Techniques, challenges, and future directions." In: *2016 International Conference on Computation of Power, Energy Information and Commuincation (ICCPEIC)*. 2016, pp. 204–221.

[AN04] Ahmed M. Ayad and Jeffrey F. Naughton. "Static Optimization of Conjunctive Queries with Sliding Windows over Infinite Streams." In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD '04. Paris, France: ACM, 2004, pp. 419–430.

[Apa] *Apache Storm*. `http://storm.apache.org`. 04.05.2021.

[Art+17] Alexander Artikis, Nikos Katzouris, Ivo Correia, Chris Baber, Natan Morar, Inna Skarbovsky, Fabiana Fournier, and Georgios Paliouras. "A Prototype for Credit Card Fraud Management: Industry Paper." In: *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*. DEBS '17. Barcelona, Spain: ACM, 2017, pp. 249–260.

[ASm09] Matthew N. Anyanwu, S. Shiva, and manyanwu. "Comparative Analysis of Serial Decision Tree Classification Algorithms." In: 2009.

[Bab+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. "Models and Issues in Data Stream Systems." In: *Proceedings of the Twenty-first ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. PODS '02. Madison, Wisconsin: ACM, 2002, pp. 1–16.

[Bal+13] Cagri Balkesen, Nihal Dindar, Matthias Wetter, and Nesime 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.

[BBS98] Sandeep Bajaj, Lee Breslau, and Scott Shenker. "Uniform versus Priority Dropping for Layered Video." In: *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*. SIGCOMM '98. Vancouver, British Columbia, Canada: Association for Computing Machinery, 1998, 131–143.

[Bel57] Richard Bellman. *Dynamic Programming*. 1957.

[Ber] "Bernoulli Distribution." In: *The Concise Encyclopedia of Statistics*. New York, NY: Springer New York, 2008, pp. 36–37.

[Bho+18] Sukanya Bhowmik, Muhammad A. Tariq, Jonas Grunert, Deepak Srinivasan, and Kurt Rothermel. "Expressive Content-Based Routing in Software-Defined Networks." In: *IEEE Transactions on Parallel and Distributed Systems* 29.11 (2018), pp. 2460–2477.

[Blo70] Burton H. Bloom. "Space/Time Trade-Offs in Hash Coding with Allowable Errors." In: *Commun. ACM* 13.7 (July 1970), 422–426.

[Bri+08] Andrey Brito, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. "Speculative Out-of-order Event Processing with Software Transaction Memory." In: *Proceedings of the Second International Conference on Distributed Event-based Systems*. DEBS '08. Rome, Italy: ACM, 2008, pp. 265–275.

[BTÖ13] Çagir Balkesen, Nesime Tatbul, and M. Tamer Özsu. "Adaptive input admission and management for parallel stream processing." In: *DEBS '13*. 2013.

[Car+02] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. "Monitoring Streams: A New Class of Data Management Applications." In: *Proceedings of the 28th International Conference on Very Large Data Bases*. VLDB '02. Hong Kong, China: VLDB Endowment, 2002, pp. 215–226.

[CF+13] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. "Integrating Scale out and Fault Tolerance in Stream Processing Using Operator State Management." In: *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*. 2013.

[CGB11] Bruno Cadonna, Johann Gamper, and Michael H. Böhlen. "Sequenced Event Set Pattern Matching." In: *Proc. of the 14th Int. Conf. on Extending Database Technology*. 2011.

[CGM10] Badrish Chandramouli, Jonathan Goldstein, and David Maier. "High-performance Dynamic Pattern Matching over Disordered Streams." In: *Proc. VLDB Endow.* 3.1-2 (2010), pp. 220–231.

[Cha+94] Sharma Chakravarthy, V. Krishnaprasad, Eman 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*. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, 606–617.

[Che+03] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Ying Xing, and Stan Zdonik. "Scalable Distributed Stream Processing." In: *CIDR*. 2003.

[CK09] Joong Hyuk Chang and Hye-Chung (Monica) Kum. "Frequency-based Load Shedding over a Data Stream of Tuples." In: *Inf. Sci.* 179.21 (2009), pp. 3733–3744.

[CLN12] Sivadon Chaisiri, Bu-Sung Lee, and Dusit Niyato. "Optimization of Resource Provisioning Cost in Cloud Computing." In: *IEEE Transactions on Services Computing* 5.2 (2012), pp. 164–177.

[CM10] Gianpaolo Cugola and Alessandro Margara. "TESLA: A Formally Defined Event Specification Language." In: *Proc. of the 4th ACM Int. Conf. on Distributed Event-Based Systems*. 2010.

[CM12] Gianpaolo Cugola and Alessandro Margara. "Processing Flows of Information: From Data Stream to Complex Event Processing." In: *ACM Comput. Surv.* 44.3 (2012), 15:1–15:62.

[CM94] Sharma Chakravarthy and Deepak Mishra. "Snoop: An Expressive Event Specification Language for Active Databases." In: *Data Knowl. Eng.* 14.1 (1994), pp. 1–26.

[Cor+04] Graham Cormode, Theodore Johnson, Flip Korn, S. Muthukrishnan, Oliver Spatscheck, and Divesh Srivastava. "Holistic UDAFs at Streaming Speeds." In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD '04. Paris, France: ACM, 2004, pp. 35–46.

[Cug+15] Gianpaolo Cugola, Alessandro Margara, Matteo Matteucci, and Giordano Tamburrelli. "Introducing uncertainty in complex event processing: model, implementation, and validation." In: *Computing* 97.2 (2015), pp. 103–144.

[DGP07] Alan Demers, Johannes Gehrke, and Biswanath P. "Cayuga: A general purpose event monitoring system." In: *In CIDR*. 2007, pp. 412–422.

[DGR03] Abhinandan Das, Johannes Gehrke, and Mirek Riedewald. "Approximate Join Processing over Data Streams." In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD '03. San Diego, California: ACM, 2003, pp. 40–51.

[DNB13] Sourav Dutta, Ankur Narang, and Suman K. Bera. "Streaming Quotient Filter: A Near Optimal Approximate Duplicate Detection Approach for Data Streams." In: *Proc. VLDB Endow.* 6.8 (2013), pp. 589–600.

[DP18] Miyuru Dayarathna and Srinath Perera. "Recent Advancements in Event Processing." In: *ACM Comput. Surv.* (2018).

[Eva96] William M. Evanco. "The impact of Rapid Incident Detection on Freeway Accident Fatalities." In: *Mitretek Report*. 1996.

[EZ+13] SF El-Zoghdy, M Nofal, MA Shohla, and AA El-sawy. "An Efficient Algorithm for Resource Allocation in Parallel and Distributed Computing Systems." In: *International Journal of Advanced Computer Science and Applications* 4 (2013).

[GD94] Stella Gatziu and Klaus R. Dittrich. "Events in an Active Object-Oriented Database System." In: *Rules in Database Systems*. London: Springer London, 1994, pp. 23–39.

[Ged+05] Buğra Gedik, Kun-Lung Wu, Philip S. Yu, and Ling Liu. "Adaptive Load Shedding for Windowed Stream Joins." In: *Proceedings of the 14th ACM International Conference on Information and Knowledge Management*. CIKM '05. Bremen, Germany: ACM, 2005, pp. 171–178.

[Ged+07] Buğra Gedik, Kun-Lung Wu, Philip S. Yu, and Ling Liu. "A Load Shedding Framework and Optimizations for M-way Windowed Stream Joins." In: *2007 IEEE 23rd International Conference on Data Engineering*. 2007, pp. 536–545.

[GFS10] Sorabh Gandhi, Luca Foschini, and Subhash Suri. "Space-efficient online approximation of time series data: Streams, amnesia, and out-of-order." In: *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*. IEEE. 2010, pp. 924–935.

[Gil+01] Anna C. Gilbert, Yannis Kotidis, S. Muthukrishnan, and Martin J. Strauss. "Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries." In: *Proceedings of the 27th International Conference on Very Large Data Bases*. VLDB '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 79–88.

[GJS92] Narain H. Gehani, H. V. Jagadish, and Oded Shmueli. "Composite Event Specification in Active Databases: Model and Implementation." In: *Proceedings of the 18th International Conference on Very Large Data Bases*. VLDB '92. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992, 327–338.

[Goo] *Google Finance*. `https://www.google.com/finance`. 05.05.2019.

[Gro+16] Michael Grossniklaus, David Maier, James Miller, Sharmadha Moorthy, and Kristin Tufte. "Frames: Data-driven Windows." In: *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*. DEBS '16. Irvine, California: ACM, 2016, pp. 13–24.

[GWY08] Buğra Gedik, Kun-Lung Wu, and Philip S. Yu. "Efficient Construction of Compact Shedding Filters for Data Stream Processing." In: *2008 IEEE 24th International Conference on Data Engineering*. 2008, pp. 396–405.

[HBN14] Yeye He, Siddharth Barman, and Jeffrey F. Naughton. "On Load Shedding in Complex Event Processing." In: *ICDT*. 2014.

[Hir+14] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. "A Catalog of Stream Processing Optimizations." In: *ACM Comput. Surv.* 46.4 (2014), 46:1–46:34.

[Hir12] Martin Hirzel. "Partition and Compose: Parallel Complex Event Processing." In: *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. DEBS '12. Berlin, Germany: ACM, 2012, pp. 191–200.

[How12] Ronald A. Howard. *Dynamic Probabilistic Systems: Markov Models*. Dover Books on Mathematics. Dover Publications, 2012.

[How13] Ronald A. Howard. *Dynamic Probabilistic Systems: Semi-Markov and Decision Processes*. Dover Books on Mathematics. Dover Publications, 2013.

[Jac88] Van L Jacobson. "Congestion Avoidance and Control." In: *Symposium Proceedings on Communications Architectures and Protocols*. SIGCOMM '88. Stanford, California, USA: Association for Computing Machinery, 1988, 314–329.

[JMR05] Theodore Johnson, S. Muthukrishnan, and Irina Rozenbaum. "Sampling Algorithms in a Stream Operator." In: *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*. SIGMOD '05. Baltimore, Maryland: ACM, 2005, pp. 1–12.

[KKL10] Hideyuki Kawashima, Hiroyuki Kitagawa, and Xin Li. "Complex Event Processing over Uncertain Data Streams." In: *2010 International Conference on P2P, Parallel, Grid, Cloud and Internet Computing*. 2010, pp. 521–526.

[KKP07] Sotiris Kotsiantis, Dimitris Kanellopoulos, and P. E. Pintelas. "Data Preprocessing for Supervised Leaning." In: *World Academy of Science, Engineering and Technology, International Journal of Computer, Electrical, Automation, Control and Information Engineering* 1 (2007), pp. 4104–4109.

[KLC18] Nikos R. Katsipoulakis, Alexandros Labrinidis, and Panos K. Chrysanthis. "Concept-Driven Load Shedding: Reducing Size and Error of Voluminous and Variable Data Streams." In: *2018 IEEE International Conference on Big Data (Big Data)*. 2018, pp. 418–427.

[Knu98] Donald E. Knuth. *The Art of Computer Programming: Volume 3: Sorting and Searching*. Pearson Education, 1998.

[Kol+12] Boris Koldehofe, Beate Ottenwälder, Kurt Rothermel, and Umakishore Ramachandran. "Moving Range Queries in Distributed Complex Event Processing." In: *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*. DEBS '12. Berlin, Germany: ACM, 2012, pp. 201–212.

[KSP14] Alok G. Kumbhare, Yogesh Simmhan, and Viktor K. Prasanna. "PLAStiCC: Predictive Look-Ahead Scheduling for Continuous Dataflows on Clouds." In: *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*. 2014, pp. 344–353.

[LG15] Zheng Li and Tingjian Ge. "PIE: Approximate interleaving event matching over sequences." In: *2015 IEEE 31st International Conference on Data Engineering*. 2015, pp. 747–758.

[LG16] Zheng Li and Tingjian Ge. "History is a Mirror to the Future: Best-effort Approximate Complex Event Matching with Insufficient Resources." In: *Proc. VLDB Endow.* 10.4 (2016), pp. 397–408.

[Li+05] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. "No Pane, No Gain: Efficient Evaluation of Sliding-window Aggregates over Data Streams." In: *SIGMOD Rec.* 34.1 (2005), pp. 39–44.

[Li+08] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. "Out-of-order Processing: A New Architecture for High-performance Stream Systems." In: *Proc. VLDB Endow.* 1.1 (2008), pp. 274–288.

[Liu+09] Mo Liu, Ming Li, Denis Golovnya, Elke A. Rundensteiner, and Kajal Claypool. "Sequence Pattern Query Processing over Out-of-Order Event Streams." In: *2009 IEEE 25th International Conference on Data Engineering*. 2009, pp. 784–795.

[LM04] Iosif Lazaridis and Sharad Mehrotra. "Approximate selection queries over imprecise data." In: *Proceedings. 20th International Conference on Data Engineering*. 2004, pp. 140–151.

[Luc01] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[MAEA05] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. "Duplicate Detection in Click Streams." In: *Proceedings of the 14th International Conference on World Wide Web*. WWW '05. Chiba, Japan: ACM, 2005, pp. 12–21.

[Man+18] Ioannis Manousakis, Íñigo Goiri, Ricardo Bianchini, Sandro Rigo, and Thu D. Nguyen. "Uncertainty Propagation in Data Processing Systems." In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC '18. Carlsbad, CA, USA: ACM, 2018, pp. 95–106.

[May+17] Ruben Mayer, Ahmad Slo, Muhammad Adnan Tariq, Kurt Rothermel, Manuel Gräber, and Umakishore Ramachandran. "SPECTRE: Supporting Consumption Policies in Window-based Parallel Complex Event Processing." In: *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. Middleware '17. Las Vegas, Nevada: ACM, 2017, pp. 161–173.

[May18] Ruben Mayer. "Window-based data parallelization in complex event processing." PhD thesis. University of Stuttgart, 2018.

[McC+13] Richard McCreadie, Craig Macdonald, Iadh Ounis, Miles Osborne, and Sasa Petrovic. "Scalable distributed event detection for Twitter." In: *Big Data, 2013 IEEE International Conference on*. 2013, pp. 543–549.

[MKR15] Ruben Mayer, Boris Koldehofe, and Kurt Rothermel. "predictable low-latency event detection with parallel complex event processing." In: *IEEE Internet of Things Journal* 2.4 (2015), pp. 274 –286.

[MM02] Gurmeet Singh Manku and Rajeev Motwani. "Approximate Frequency Counts over Data Streams." In: *Proceedings of the 28th International Conference on Very Large Data Bases*. VLDB '02. Hong Kong, China: VLDB Endowment, 2002, pp. 346–357.

[MM09] Yuan Mei and Samuel Madden. "ZStream: a cost-based query processor for adaptively detecting composite events." In: *SIGMOD Conference*. 2009.

[MM16] Tiziano D. Matteis and Gabriele Mencagli. "Parallel Patterns for Window-Based Stateful Operators on Data Streams: An Algorithmic Skeleton Approach." In: *International Journal of Parallel Programming* 45 (2016), pp. 382–401.

[Mot+03] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. "Query Processing, Approximation, and Resource Management in a Data Stream Management System." In: *CIDR*. 2003.

[MP13] Christopher Mutschler and Michael Philippsen. "Reliable Speculative Processing of Out-of-order Event Streams in Generic Publish/Subscribe Middlewares." In: *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*. DEBS '13. Arlington, Texas, USA: ACM, 2013, pp. 147–158.

[MTR17] Ruben Mayer, Muhammad Adnan Tariq, and Kurt Rothermel. "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. Barcelona, Spain: ACM, 2017, pp. 54–65.

[MZJ13] Christopher Mutschler, Holger Ziekow, and Zbigniew Jerzak. "The DEBS 2013 Grand Challenge." In: *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*. DEBS '13. Arlington, Texas, USA: ACM, 2013, pp. 289–294.

[Neu+10] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. "S4: Distributed Stream Computing Platform." In: *Data Mining Workshops (ICDMW), IEEE Int. Conf.* 2010.

[OJW03] Chris Olston, Jing Jiang, and Jennifer Widom. "Adaptive Filters for Continuous Queries over Distributed Data Streams." In: *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*. 2003.

[PS06] Kostas Patroumpas and Timos Sellis. "Window Specification over Data Streams." In: *Proceedings of the 2006 International Conference on Current Trends in Database Technology*. EDBT'06. Munich, Germany: Springer-Verlag, 2006, pp. 445–464.

[Qui93] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993. ISBN: 1558602380.

[Quo+17] Do Le Quoc, Ruichuan Chen, Pramod Bhatotia, Christof Fetzer, Volker Hilt, and Thorsten Strufe. "StreamApprox: Approximate Computing for Stream Analytics." In: *Proc. of the 18th ACM/IFIP/USENIX Middleware Conf.* 2017.

[RBQ16] Nicoló Rivetti, Yann Busnel, and Leonardo Querzoni. "Load-aware Shedding in Stream Processing Systems." In: *Proc. of the 10th ACM Int. Conf. on Distributed and Event-based Systems*. 2016.

[RBR19] Henriette Röger, Sukanya Bhowmik, and Kurt Rothermel. "Combining It All: Cost Minimal and Low-Latency Stream Processing across Distributed Heterogeneous Infrastructures." In: *Proceedings of the 20th International Middleware Conference*. Middleware '19. Davis, CA, USA: Association for Computing Machinery, 2019, 255–267.

[RH05] Frederick Reiss and Joseph M. Hellerstein. "Data Triage: an adaptive architecture for load shedding in TelegraphCQ." In: *21st International Conference on Data Engineering (ICDE'05)*. 2005, pp. 155–156.

[Riv+18] Nicolo Rivetti, Nikos Zacheilas, Avigdor Gal, and Vana Kalogeraki. "Probabilistic Management of Late Arrival of Events." In: *Proceedings of the 12th ACM International Conference on Distributed and Event-based Systems*. DEBS '18. Hamilton, New Zealand: ACM, 2018, pp. 52–63.

[RLR16] Medhabi Ray, Chuan Lei, and Elke A. Rundensteiner. "Scalable Pattern Sharing on Event Streams." In: *Proc. of the Int. Conf. on Management of Data*. 2016.

[RM19] Henriette Röger and Ruben Mayer. "A Comprehensive Survey on Parallelization and Elasticity in Stream Processing." In: *ACM Comput. Surv.* 52.2 (2019).

[Sad+04] Reza Sadri, Carlo Zaniolo, Amir Zarkesh, and Jafar Adibi. "Expressing and Optimizing Sequence Queries in Database Systems." In: *ACM Trans. Database Syst.* 29.2 (2004), 282–318.

[SBR19] Ahmad Slo, Sukanya Bhowmik, and Kurt Rothermel. "eSPICE: Probabilistic Load Shedding from Input Event Streams in Complex Event Processing." In: *Proceedings of the 20th ACM/IFIP Middleware Conference*. Middleware '19. UC Davis, CA, USA: ACM, 2019.

[SBR20a] Ahmad Slo, Sukanya Bhowmik, and Kurt Rothermel. "HSPICE: State-Aware Event Shedding in Complex Event Processing." In: *Proceedings of the 14th ACM International Conference on Distributed and Event-Based Systems*. DEBS '20. Montreal, Quebec, Canada: ACM, 2020, 109–120.

[SBR20b] Ahmad Slo, Sukanya Bhowmik, and Kurt Rothermel. "State-Aware Load Shedding from Input Event Streams in Complex Event Processing." In: *IEEE Transactions on Big Data* 01 (2020), pp. 1–1.

[Sha+03] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. "Flux: an adaptive partitioning operator for continuous query systems." In: *Data Engineering, 2003. Proceedings. 19th International Conference on*. 2003, pp. 25–36.

[Ski08] Steven S. Skiena. "Sorting and Searching." In: *The Algorithm Design Manual*. London: Springer London, 2008, pp. 103–144.

[Slo+19] Ahmad Slo, Sukanya Bhowmik, Albert Flaig, and Kurt Rothermel. "pSPICE: Partial Match Shedding for Complex Event Processing." In: *2019 IEEE International Conference on Big Data*. Los Angeles, CA, USA, 2019.

[SMMP09] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. "Distributed Complex Event Processing with Query Rewriting." In: *Proc. of the 3rd ACM Int. Conf. on Distributed Event-Based Systems*. 2009.

[SW04] Utkarsh Srivastava and Jennifer Widom. "Memory-limited Execution of Windowed Stream Joins." In: *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*. VLDB '04. Toronto, Canada: VLDB Endowment, 2004, pp. 324–335.

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

[TBL08] Wee Hyong Tok, Stéphane Bressan, and Mong-Li Lee. "A Stratified Approach to Progressive Approximate Joins." In: *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*. EDBT '08. Nantes, France: ACM, 2008, pp. 582–593.

[TcZ07] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. "Staying FIT: Efficient Load Shedding Techniques for Distributed Stream Processing." In: *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB '07. Vienna, Austria: VLDB Endowment, 2007, pp. 159–170.

[TZ06] Nesime Tatbul and Stan Zdonik. "Window-aware Load Shedding for Aggregation Queries over Data Streams." In: *Proceedings of the 32Nd International Conference on Very Large Data Bases*. VLDB '06. Seoul, Korea: VLDB Endowment, 2006, pp. 799–810.

[Upd18] 360 Market Updates. *Global complex event processing market.* Accessed: 2020-08-27. 2018. URL: `https://www.360marketupdates.com/global-complex-event-processing-market-12886110`.

[Wal+99] Jonathan Walpole, Ling Liu, David Maier, Calton Pu, and Charles Krasic. "Quality of Service Semantics for Multimedia Database Systems." In: 1999, pp. 393–412.

[WDR06] Eugene Wu, Yanlei Diao, and Shariq Rizvi. "High-performance Complex Event Processing over Streams." In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. Chicago, IL, USA: ACM, 2006, pp. 407–418.

[WRM10] Mingzhu Wei, Elke A. Rundensteiner, and Murali Mani. "Achieving High Output Quality Under Limited Resources Through Structure-based Spilling in XML Streams." In: *Proc. VLDB Endow.* 3.1-2 (2010), pp. 1267–1278.

[WTA10] Louis Woods, Jens Teubner, and Gustavo Alonso. "Complex Event Detection at Wire Speed with FPGAs." In: *Proc. VLDB Endow.* (2010).

[YBT05] Jia Yu, Rajkumar Buyya, and Chen Khong Tham. "Cost-based scheduling of scientific workflow applications on utility grids." In: *e-Science and Grid Computing, 2005. First International Conference on*. 2005, 8 pp.–147.

[Zac+15] Nikos Zacheilas, Vana Kalogeraki, Nikolas Zygouras, Nikolaos Panagiotou, and Dimitrios Gunopulos. "Elastic complex event processing exploiting prediction." In: *IEEE Int. Conf. on Big Data*. 2015.

[ZDI10] Haopeng Zhang, Yanlei Diao, and Neil Immerman. "Recognizing Patterns in Streams with Imprecise Timestamps." In: *Proc. VLDB Endow.* 3.1–2 (2010), 244–255.

[ZDI14] Haopeng Zhang, Yanlei Diao, and Neil Immerman. "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. Snowbird, Utah, USA: ACM, 2014, pp. 217–228.

[Zob90] Albert L. Zobrist. "A New Hashing Method with Application for Game Playing." In: *ICGA Journal* 13 (1990), pp. 69–73.

[ZR10] Erik Zeitler and Tore Risch. "Massive scale-out of expensive continuous queries." In: *36th Int. Conf. on Very Large Data Bases : VLDB 2010*. 2010.

[ZU99] Detlef Zimmer and Rainer Unland. "On the Semantics of Complex Events in Active Database Management Systems." In: *Proceedings of the 15th International Conference on Data Engineering*. ICDE '99. Washington, DC, USA: IEEE Computer Society, 1999.

[ZVHW20] Bo Zhao, Nguyen Quoc Viet Hung, and Matthias Weidlich. "Load Shedding for Complex Event Processing: Input-based and State-based Techniques." In: *ICDE 2020*. 2020.

[Jai90] Raj Jain. "Congestion control in computer networks: issues and trends." In: *IEEE Network* 4.3 (1990), pp. 24–30.

[Let+10] Julie Letchner, Christopher Ré, Magdalena Balazinska, and Matthai Philipose. "Approximation trade-offs in Markovian stream processing: An empirical study." In: *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. 2010, pp. 936–939.

[Lim+18] Guilherme F. Lima, Ahmad Slo, Sukanya Bhowmik, Markus Endler, and Kurt 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)*. 2018, pp. 31–40.

[Mor+19] Nathalia Moreno, Manuel F. Bertoa, Loli Burgueño, and Antonio Vallecillo. "Managing Measurement and Occurrence Uncertainty in Complex Event Processing Systems." In: *IEEE Access* 7 (2019), pp. 88026–88048.

[Tin95] Tin Kam Ho. "Random decision forests." In: *Proceedings of 3rd International Conference on Document Analysis and Recognition*. Vol. 1. 1995, 278–282 vol.1.

Erklärung

Ich erkläre hiermit, dass ich, abgesehen von den ausdrücklich bezeichneten Hilfsmitteln und den Ratschlägen von jeweils namentlich aufgeführten Personen, die Dissertation selbstständig verfasst habe.

(Ahmad Slo)