DARLING: Data-Aware Load Shedding in Complex Event Processing Systems

Koral Chapnik
Technion, Israel Institute of Technology
Haifa, Israel
skoralch@campus.technion.ac.il

Ilya Kolchinsky
Technion, Israel Institute of Technology
Haifa, Israel
ikolchin@cs.technion.ac.il

Assaf Schuster
Technion, Israel Institute of Technology
Haifa, Israel
assaf@technion.ac.il

ABSTRACT

Complex event processing (CEP) is widely employed to detect user-defined combinations, or patterns, of events in massive streams of incoming data. Numerous applications such as healthcare, fraud detection, and more, use CEP technologies to capture critical alerts, threats, or vital notifications. This requires that the technology meet real-time detection constraints. Multiple optimization techniques have been developed to minimize the processing time for CEP, including parallelization techniques, pattern rewriting, and more. However, these techniques may not suffice or may not be applicable when an unpredictable peak in the input event stream exceeds the system capacity. In such cases, one immediate possible solution is to drop some of the load in a technique known as load shedding.

We present a novel load shedding mechanism for real-time complex event processing. Our approach uses statistics that are gathered to detect overload. The solution makes data-driven load shedding decisions to drop the less important events such that we preserve a given latency bound while minimizing the degradation in the quality of results. An extensive experimental evaluation on a broad set of real-life patterns and datasets demonstrates the superiority of our approach over the state-of-the-art techniques.

PVLDB Reference Format:
Koral Chapnik, Ilya Kolchinsky, and Assaf Schuster. DARLING: Data-Aware Load Shedding in Complex Event Processing Systems. PVLDB, 15(3): 541 - 554, 2022.
doi:10.14778/3494124.3494137

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/koralchapnik/darling.git.

1 INTRODUCTION

The discovery of complex combinations, or *patterns*, in data items can be used to capture critical alerts, potential threats, vital notifications, or rare and unique opportunities. Complex event processing (CEP) is a technology aimed at the *efficient real-time* detection of such complex data patterns over massive input data streams. Online finance [6, 52], fraud detection [11, 54], and security monitoring are among the many applications employing CEP technologies. Figure 1 illustrates the use of CEP in the fraud detection domain.

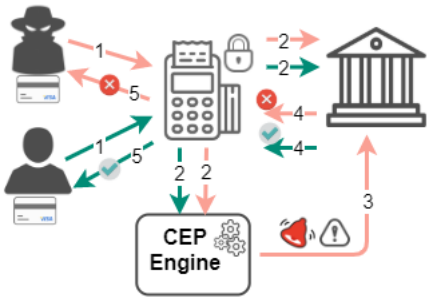


Figure 1: Fraud detection process. Clients attempt to perform transactions using a credit card (1). The data is sent to the card's issuer bank and the CEP system (2). The CEP system detects a fraud attempt and immediately alerts the bank (3), which refuses the transaction (4,5).

In fraud detection, for example, online data items of credit card usage continuously stream into a CEP engine from one or more data sources. Such data items arriving from input event streams are referred to as *primitive events*, or simply *events*. Each event belongs to an *event type*, which defines a set of attributes to be associated with an event. For example, one could defined `BigTransReq` as a type representing a transaction request for a large sum; its attributes could then be $(cardID, amount)$.

CEP patterns specify scenarios of interest to be detected. For example, assume we want to detect a potential fraud attempt by recognizing a sequence of two failed CVV attempts, followed by a successful CVV attempt. In fraud attempts, this may typically be followed by an initial small transaction request and then a second large transaction request for an amount greater than \$10,000, and at least 10 times greater than the preceding small transaction request amount. All this occurs within a time window of five minutes. Such a pattern could be formally written as follows:

$$PATTERN(1)\ SEQ(CvvRefused1\ a,\ CvvRefused2\ b,$$
$$CvvConfirmed\ c,\ SmallTransReq\ d, BigTransReq\ e)$$
$$WHERE\ (e.amount > d.amount \times 10)\ \wedge (e.amount > 10,000)$$
$$\wedge (a.cardID = b.cardID = c.cardID = d.cardID = e.cardID)$$
$$WITHIN\ 5\ minutes$$

Detecting pattern matches while maintaining low latency is crucial in a variety of application domains. In our example above, a failure to quickly detect a fraud attempt could allow a fraudster to

Proceedings of the VLDB Endowment, Vol. 15, No. 3 ISSN 2150-8097.
doi:10.14778/3494124.3494137

use the credit card of the victim. The maximum acceptable detection *latency bound* in this case is the time it takes from the large transaction request until the card issuing bank responds to the request. Fraud alerts received after the transaction approval will be useless as the money has already been withdrawn.

Many optimization techniques have been proposed to minimize the processing latency in CEP systems [16, 19, 20, 28, 40, 55, 56], including pattern rewriting, sub-expression sharing, and parallelization. However, a sudden and unpredictable peak in the rate of incoming events may exceed the system capacity, rendering the above optimizations useless and leading to inevitable data loss. In these cases, an immediate action must be taken to minimize this data loss and ensure that the remaining pattern matches are still detected within the required latency bound.

One possible solution for the above situation could be to dynamically scale up the application or to overprovision it in advance. However, with the recent advances in edge processing and IoT, CEP is often handled at the edge by a multitude of small and inexpensive devices with restricted resources. Therefore, the aforementioned approach would not be applicable in many common use cases where scaling up or over-provisioning resources is not possible.

In this paper, we discuss an alternative solution, which is to drop some of the load. This process is also known as *load shedding*. Dropping load prevents the formation of large queues and keeps the detection latency below the desired latency bound. While some fraction of the data is still lost, correctly performed load shedding attempts to minimize the loss of 'important' data items. Achieving the perfect balance between maintaining the latency bound and minimizing this loss is the main challenge in practically applying load shedding. In the fraud detection example, load must be shed in a way that maintains as many fraud alerts as possible. This can be achieved by dropping the events that are less likely to be part of a pattern match such as Pattern (1). Additionally, the load shedding process must incur minimal overhead.

Load shedding has gained much attention in the area of classic data stream processing [30, 31, 38, 39, 41, 44, 51, 53]. However, CEP imposes new difficulties and amplifies the existing challenges in producing optimal load shedding decisions. CEP systems must capture the correlations and dependencies created by combining multiple data items. For example, if we revisit Pattern (1), dropping all events indicating an incorrect CVV code (CvvRefused event types) will result in zero detected matches. In general, the contribution of each event to the overall load and to the system output depends on the combination of multiple parameters.

Load shedding in CEP has been studied by numerous authors [27, 47–49, 58]. However, these works share a number of limitations. First, most authors assume a strong connection between an event's importance and either its position in the window [48, 49], the importance of a partial match containing it [58], or its frequency of appearance [27]. These assumptions do not always hold. In addition, all published methods except [58] lack a semantic load shedding mechanism, and all existing works that use a cost model based on partial matches [47, 49, 58] perform poorly in real-life CEP settings, as we experimentally evaluate in this paper.

We present a novel, scalable, and efficient load shedding mechanism for CEP systems. DARLING (DAta dRiven Load sheddING) partitions the input event stream into dedicated buffers from which the CEP engine consumes events. Considering arrival rates, cross-buffer correlations, and estimated processing complexity, DARLING assigns a global constraint on the size of the input event stream and local constraints on the size of each of these buffers. The former is utilized to detect overload situations, and the latter constraints dictate from which buffers to drop events and how much to drop. In addition, DARLING estimates the importance of each event with a utility function computed via data-driven statistical methods, which allows it to drop the least important events first.

We conducted an extensive experimental study comparing DARLING to three state-of-the-art CEP load shedding methods. The experiments, covering three real-world datasets and a broad set of patterns, demonstrated the scalability of our approach and its superiority over the examined methods in terms of latency, percentage of detected matches, and processing overhead.

2 BACKGROUND AND TERMINOLOGY

2.1 Complex Event Processing

The goal of a complex event processing (CEP) system is to monitor massive high-speed data streams for combinations of events satisfying user-specified *patterns*.

A pattern is defined by a combination of primitive events, operators, predicates, and a time window. The most commonly used operators in CEP are sequence, conjunction, disjunction, negation (i.e., absence of an event from some position in the match) and Kleene closure (i.e., one or more instances of an event).

As an example, the following pattern from the fraud detection domain consists of a sequence of three event types that represent small, medium, and large transaction requests:

$$PATTERN(2)\ SEQ(SmallTrans\ a,\ MediumTrans\ b,\ BigTrans\ c)$$
$$WHERE(a.cardID = b.cardID = c.cardID) \wedge (a.amount > 100)$$
$$\wedge (b.amount > a.amount \times 10) \wedge (c.amount > b.amount \times 100)$$
$$WITHIN\ 5\ minutes$$

CEP engines use an *evaluation mechanism* to create an internal representation of the pattern, defining the loose order by which events are combined during the pattern matching process. The most common evaluation mechanisms are trees [37], non-deterministic finite automata [55], and event processing networks [21].

In a tree evaluation mechanism, each leaf represents an event type. For clarity, we denote the leaf name as the corresponding event type name. Primitive events continuously stream into the system, arrive at the appropriate leaf, and are processed up the tree. Each node in the tree contains a set of conditions to be satisfied by the combinations of incoming events, including time window constraints and pattern predicates. When all conditions are satisfied by some combination, it is stored at the node as a *partial match* (PM) and transferred up the tree where the matching procedure repeats. Partial matches that reach the tree root and satisfy its conditions become *full matches* and are reported to the user.

Figure 2 displays a tree evaluation process for Pattern 2. Assume all events have the same cardID attribute. When event a_1 arrives in the system, it is routed to the corresponding leaf *Small*, and tested against its conditions. Since $a_1.amount > 100$, a new partial match $PM1$ is created. When b_1 arrives at *Medium* leaf, since there are no conditions to be satisfied, a new partial match $PM2$ is created

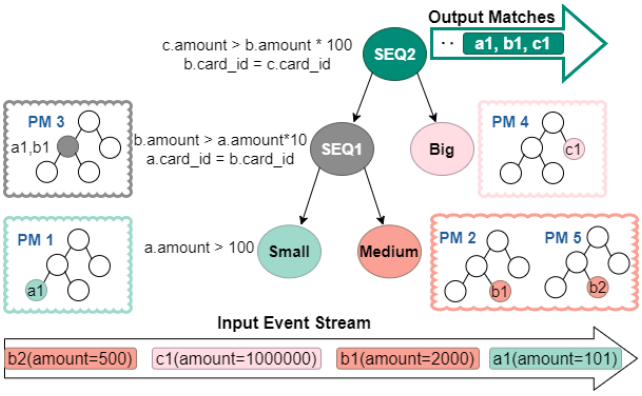


Figure 2: Example of tree evaluation mechanism. Events from the input event stream are routed to the corresponding leaves and processed up the tree, creating combinations of events that form PMs when satisfying the node conditions.

and saved in the node. $PM2$ is then combined with $PM1$ to form a combination (a_1, b_1), which is tested against the conditions in the parent node $SEQ1$. Since the combination satisfies the conditions, it forms a new partial match $PM3$, at $SEQ1$. At the moment, no events of type $BigTrans$ have arrived; hence, there are no partial matches to be combined with $PM3$ and the processing of b_1 terminates. When an event c_1 is processed, it creates $PM4$, which is then combined with $PM3$. This new combination satisfies the root conditions, creating a match (a_1, b_1, c_1) that is reported to the user.

The number of partial matches created during evaluation is influenced by numerous factors. One major factor is the distribution of event attribute values. Revisiting Figure 2, when a new event b_2 arrives in the system and creates $PM5$, $PM5$ is combined with $PM1$. However, no partial match is formed as $b_2.amount \not> a_1.amount \times 10$. The event attributes essentially impact the *predicate selectivity* - the probability that a combination of events will pass the predicate.

Additional factors directly proportional to the number of PMs created are the window size and the arrival rates of the different event types. The window size determines the maximum possible time difference between two events in a match. The event type *arrival rate* specifies the number of primitive events of this type that arrive in the system per time unit.

Each newly arrived event is evaluated against all the relevant PMs in the system. Thus, the more PMs are stored, the longer it takes to process an event, thereby reducing the detection latency.

2.2 Load Shedding

As we defined in Section 1, load shedding is a technique used to reduce latencies in overload situations under a resource-constrained environment. In general terms, load shedding is performed by dropping objects from the system. While load shedding is vital for maintaining the latency bound, dropping objects can have a drastic effect on the quality of results. Therefore, a load shedding algorithm must quickly detect situations where the risk of violating the defined latency bound is present, shed load with minimal degradation of system output, and incur minimal performance overhead.

3 PROBLEM DEFINITION

In a system that monitors numerous patterns, the patterns can have different levels of importance. For example, in a healthcare scenario, signals from life-support systems would be much more sensitive to lost matches than those from other smart sensors. Moreover, in multi-user systems, some of the users may be eligible for higher quality-of-service (e.g., premium member) and thus their patterns are considered more important than those of regular users. We encode these a-priori defined priorities in the form of weights.

We formalize the load shedding problem in CEP as the following optimization problem. Given an evaluation mechanism χ, a set of n input event streams $S = \{S_1, ..., S_n\}$, a set of patterns to be detected $P = \{p_1, ..., p_k\}$, their corresponding weights $W = \{w_1, ..., w_k\}$, a bound Λ for the metric to be maintained, and a predicate λ to check if the metric is below the allowed bound, the goal is to find a load shedding decision function $\psi : O \rightarrow \{True, False\}$ that receives an event and returns an indication to either shed it ($True$) or keep it in the system for further processing ($False$). We denote ψ_0 as a decision function that never drops an event.

Let $\lambda : (P, S, \chi, \psi) \rightarrow \{True, False\}$ be a predicate that measures the metric to be maintained while the patterns in P are monitored over S using χ with the decision function ψ. The predicate returns $True$ if the metric was maintained below the defined bound Λ during the system run-time and $False$ otherwise.

As explained in Sections 1 and 2, load shedding drops objects from the system and consequently affects the system output. Assume $SCORE : (P, W, S, \chi, \psi) \rightarrow \mathbb{R}$ is a function that represents the score of the system, namely, how well the system performs.

Using the above notations, we formally define the load shedding problem in CEP as follows:

$$\begin{aligned} &ARGMAX_{\psi}\ SCORE(P, W, S, \chi, \psi) \\ &\quad s.t. \quad \lambda(P, S, \chi, \psi) = True \end{aligned} \tag{1}$$

Our goal is to keep the detection latency of a match below a user-defined latency bound $\Lambda = L_{max}$. The detection latency $l(m)$ of match m, is defined as the time it takes from the arrival of the latest event to complete the match m, until m is detected by the system. Specifically, if $t_{det}(m)$ is the time when the system detected the match m, and $e.timestamp$ is the arrival time of event e, then:

$$l(m) = t_{det}(m) - max\ \{e.timestamp\ \forall e \in m\} \tag{2}$$

We define $M(P, S, \chi, \psi)$ as the set of matches detected by the system for all the patterns when applying the load shedding decision function ψ, and $M_{p_i}(P, S, \chi, \psi)$ as the set of matches of pattern p_i that were detected by the system. Then, the predicate λ will indicate whether the detection latency is preserved:

$$\lambda(P, S, \chi, \psi) = \begin{cases} True & \forall m \in M(P, S, \chi, \psi) : (l(m) \leq L_{max}) \\ False & otherwise \end{cases} \tag{3}$$

The system score reflects the rate of detected matches associated with the patterns' importance, which is encoded in the form of weights. Therefore, we represent the system score as:

$$SCORE(P, W, S, \chi, \psi) = \frac{\sum_{i=1}^{i=k} |M_{p_i}(P, S, \chi, \psi) \cap M_{p_i}(P, S, \chi, \psi_0)| w_i}{\sum_{i=1}^{i=k} |M_{p_i}(P, S, \chi, \psi_0)| w_i} \tag{4}$$

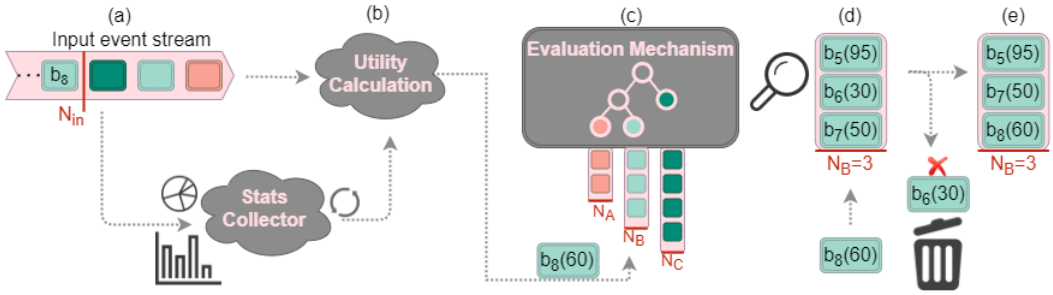


Figure 3: Load shedding process in DARLING. As new event b_8 arrives in the system and the global constraint N_{in} is violated (a), the event's utility value is calculated using the statistics gathered (b). The system then routes the event to the corresponding buffer of event type B (c). Since buffer B is full and its local constraint is violated, the event b_6 with the lowest utility value is dropped from the system (d), and b_8 is added to the buffer (e) for further processing.

The intersection in the numerator is necessary to avoid considering false positive matches that do not appear in $M_{p_i}(P, S, \chi, \psi_0)$. Such matches can be generated, for example, when events that appear under the negation operator are dropped. The negation operator essentially requires the absence of an event from some position in the match. Therefore, dropping this event can create a false positive match if the other conditions of the pattern are satisfied.

4 DARLING

4.1 Load Shedding in DARLING

DARLING addresses three integral components of load shedding: (1) detecting overload, (2) deciding how much to drop, and (3) deciding which objects to drop.

At periodic intervals, when it has enough resources available, DARLING gathers statistics on the distribution of attribute values, event-type arrival rates, and selectivity of the conditions.

Using these statistics, DARLING computes N_{in}, the maximum number of queued events in the input event stream before latencies exceed L_{max}. The value N_{in} is a global constraint for the size of the input event stream, and indicates the need to drop events (1); its calculation is detailed in Section 4.2.1. For each event type, DARLING creates a buffer that accumulates events of the corresponding type for further processing by the evaluation mechanism. Taking into consideration the effect of each event type on the system score, and the correlations between the different event types, DARLING splits N_{in} between the different event types buffers, such that the buffer of event type T will have a maximum size of N_T, and $\sum_T N_T = N_{in}$. The values of N_T are used as local constraints for the buffer sizes to determine which buffer will drop events, and how much to drop (2). The calculations of N_T are explained in Sections 4.2.2 and 4.2.3.

When the global constraint is violated, the local constraints are verified. If the local constraint of a buffer is violated, events must be dropped from this buffer. When events are dropped, this essentially prevents them from being used to detect matches that contain them, thereby reducing the number of detected matches. To minimize the degradation in the rate of detected matches, DARLING drops only the less important events (3). To determine which events are less important for the detected matches, DARLING calculates a utility value per event. The higher the utility value of an event, the more important it is. These values are calculated on-the-fly using utility functions that were created in the statistics gathering phase. The calculation of utilities is explained in Section 4.3.

Dropping the events with the lowest utility from the buffer requires that we maintain sorted buffers, which is time-consuming. To perform fast load shedding decisions, we introduce new auxiliary data structures for each event type buffer, with negligible memory overhead. We describe these data structures in Section 4.4.

Figure 3 illustrates the flow of DARLING in overload situations where the global constraint N_{in} is violated. In non-overload mode, when N_{in} is not exceeded, utility values are not calculated and events are simply appended to their corresponding buffers.

To account for data drift, the statistics gathering phase is activated upon concept drift detection [24, 25, 31]. The values of N_{in}, N_T, and the utility values are updated accordingly, and any excess load dropped is based on the new utility calculations.

4.2 Setting Global and Local Constraints

DARLING sets a global constraint N_{in} on the size of the queued events from the input event stream; an overload situation is indicated when its value is exceeded. Furthermore, DARLING creates a separate buffer for each event type T with a local constraint N_T to indicate the maximum buffer size in overload situations.

Next, we explain how to set the global and local constraints to preserve detection latencies, as defined in Equation (2) under a given latency bound L_{max}.

In general, latencies grow in response to high resource utilization. More specifically, in CEP, computing complicated predicates over a potentially exponential number of partial matches can overload the processor and cause latencies to increase.

For simplicity, the following calculations assume an underlying tree evaluation mechanism χ for a single pattern. To simplify our explanation, we assume the pattern is purely conjunctive, containing an AND operator and a conjunction between the pattern conditions. This follows previous work [33] that describes how a pattern or nested patterns containing SEQ, OR, NOT, and Kleene closure operators can be represented and detected as either a pure conjunctive pattern or their union. Potential extensions of DARLING to multiple patterns and additional evaluation mechanisms are described in Section 4.5.

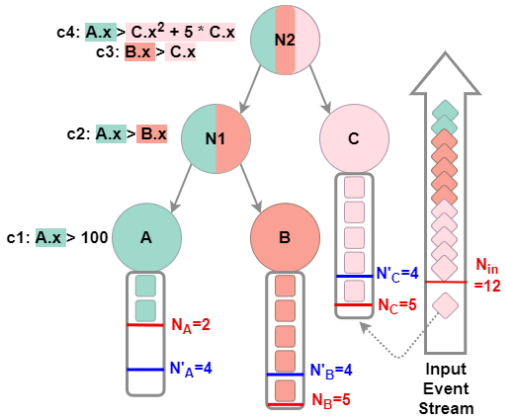


Figure 4: Example for poor split of N_{in} to N'_T

4.2.1 Calculating N_{in}. Assume that $\bar{z}$ is the average time used to process a single event in a steady state of the system, and is calculated during the system run-time.

Now, assume w.l.o.g. a match m whose latest event is e_n, namely, $argmax_e \{e.timestamp \ \forall e \in m\} = e_n$, and that e_n is the n-th event queued in the input event stream. The detection latency of m, $l(m)$ is measured from the moment e_n arrives at the system until the system has finished processing it. Essentially, this is the amount of time it takes to process all $n-1$ events in the input event stream that precede e_n's arrival, plus the time to process event e_n. As mentioned above, in a steady state, the average time used to process an arbitrary event is $\bar{z}$, hence the time needed to process n events is $n \times \bar{z}$. To avoid exceeding L_{max}, we require that $l(m) \leq L_{max}$. Consequently, the following must hold:

$$n \times \bar{z} \leq L_{max} \Longrightarrow n \leq \frac{L_{max}}{\bar{z}} \Longrightarrow N_{in} = \frac{L_{max}}{\bar{z}}$$

4.2.2 Calculating N_T. Our motivation for not dividing N_{in} equally between the different buffers stems from the fact that processing events of different types can take different amounts of time. This time period will depend on the system state and the number of PMs that exist in the system, the complexity of the conditions, the attributes' content, and arrival rates for various event types.

In Figure 4, assume the arrival rates: $R_A = 2, R_B = 5, R_C = 5$. Moreover, assume that $N_{in} = 12$ containing 2, 5, and 5 events of types A, B, and C, respectively. If we split N_{in} equally between the buffers, each event type will have a local constraint of $N'_A = N'_B = N'_C = 4$. When the 6th event of type C arrives in the system, the global constraint is violated and the local constraints are tested to see where excess load should be dropped. In this case, the buffer of C, which contains 6 events, will drop 2 events. Clearly, this would make no sense because the buffer of A has 2 available slots. In this case, the red lines would serve as better local constraints. We next introduce a superior method for splitting N_{in} among event type buffers and minimizing the unnecessary dropping of events.

Assume that $Z(T)$ (Section 4.2.3) is the average time used to process a single arbitrary event of type T, $R(T)$ is the arrival rate of event type T, and that $TYPES$ is the set of event types in the pattern. Moreover, assume that $\beta \in [0, 1]$ indicates the proportion of the global constraint N_{in} after which we start to drop events. β is a user-defined parameter employed as a safety factor to guarantee that L_{max} is not violated. Hence, the local constraint for each buffer of event type T is:

$$N_T = \frac{Z(T) \times R(T)}{\sum_{T' \in TYPES} Z(T') \times R(T')} \times N_{in} \times \beta$$

4.2.3 Calculating $Z(T)$. We first introduce some preliminary notations. Given a tree evaluation mechanism χ and an event type T, we define $C_{type}(T)$ as the set of conditions that contains attributes of event type T, and $C_{node}(N)$ as the conditions set in node $N \in \chi$. For example, in Figure 4, $C_{type}(A) = \{c_1, c_2, c_4\}$, $C_{type}(B) = \{c_2, c_3\}$, and $C_{type}(C) = \{c_3, c_4\}$. Moreover, $C_{node}(A) = \{c_1\}$, $C_{node}(B) = C_{node}(C) = \emptyset$, $C_{node}(N_1) = \{c_2\}$, and $C_{node}(N_2) = \{c_3, c_4\}$.

Note that in a tree or DAG evaluation mechanism, each leaf represents an event type T. All conditions in $C_{type}(T)$ appear in the nodes on the path from the corresponding leaf representing T to the root, which we refer to as $PATH_\chi(T)$. In Figure 4, $PATH_\chi(A) = \{A, N_1, N_2\}$, $PATH_\chi(B) = \{B, N_1, N_2\}$, and $PATH_\chi(C) = \{C, N_2\}$.

The processing time used to calculate each condition c_i depends on the condition's complexity. For example, in Figure 4, calculating condition c_4 will take more processing time than calculating c_1. Therefore, we assume that the evaluation of each condition c_i takes f_i time units, where f_i can be easily calculated.

When a new event e_T of type T arrives in the system, it triggers the formation of new partial matches. The number of partial matches formed in each node depends on the other existing partial matches in the system. This means that the arrival of e_T creates a different number of new PMs at different nodes on $PATH_\chi(T)$. Assume that $PM(N, T)$ represents the average number of *new* partial matches formed in node N, caused by the arrival of a new event of type T. Moreover, assume that $PM(N)$ is the average number of partial matches accumulated in node N. In what follows, we show how to compute $PM(N, T)$ and $PM(N)$. Let V_n be the processing time it takes to create a new partial match (i.e., create a new object) and insert it into the system, and V_r be the processing time it takes to remove a partial match from the system.

We use the following formulas to calculate $Z(T)$, the average time it takes to process a new event of type T:

$$L_{cond}(N) = \sum_{c_i \in C_{node}(N)} f_i \tag{5}$$

$$L_{peer}(N) = L_{cond}(parent\ (N)) \times \prod_{N' \in peers(N)} PM(N') \tag{6}$$

$$Z(T) = \sum_{N \in PATH_\chi(T)} (L_{peer}\ (N) + V_n + V_r) \times PM\ (N, T) \tag{7}$$

$L_{cond}(N)$ represents the processing time needed to evaluate all the conditions of node N for a single combination of primitive events. $L_{peer}(N)$ represents the processing time needed to combine a single PM in node N with all the corresponding partial matches in the peer nodes of N ($peers(N)$), and to evaluate the parent node conditions on the newly created combinations. Equation (7) describes the total processing time used for all calculations resulting from the arrival of e_T to the system. For each node N in the path from the leaf of T to the root, we calculate the number of new partial matches created in N from the arrival of e_T to the system, $PM\ (N, T)$. For

each new partial match, we calculate the time it takes to combine it with all PMs in the peer nodes and evaluate the parent conditions, $L_{peer}(N)$, and the time to create and insert the PM in the system, and then remove it from the system when it is expired.

To complete the formulation, we only need to show how to compute $PM(N,T)$ and $PM(N)$. We start by explaining the calculation of $PM(N,T)$. Assume $PM(N)$ represents the average number of PMs existing at node N, and that $SEL(c_i)$ represents the selectivity of condition c_i. Then, the selectivity of node N is: $SEL(N) = \prod_{c_i \in C_{node}(N)} SEL(c_i)$. For a leaf node L, the average number of new PMs created in L as an event of type T arrives is $SEL(L)$, provided L is the corresponding leaf of event type T; otherwise 0:

$$PM(L,T) = SEL(L) \times \mathbb{1}_{L=T} \tag{8}$$

The number of new PMs created in node N due to the arrival of an event of type T can be calculated as:

$$\begin{aligned} PM(N,T) = SEL(N) \times & \prod_{N'' \in sons(N) \cap PATH_{\chi}(T)} PM(N'',T) \\ \times & \prod_{N' \in sons(N) \setminus PATH_{\chi}(T)} PM(N') \end{aligned} \tag{9}$$

Where $sons(N)$ represents the children of node N. The number of new combinations created is the number of new PMs in the child belonging to $PATH_{\chi}(T)$ multiplied by the existing PMs in the other children. Multiplying this number of all possible new combinations by the selectivity of N will result in the number of new PMs created in N as an event of type T arrived.

Similarly, given the arrival rate $R(T)$ of event type T and the window size w, the expected number of partial matches $PM(N)$ in node N can be calculated as follows:

$$PM(N) = \begin{cases} R(N) \times SEL(N) \times w & N \text{ is a leaf} \\ SEL(N) \times \prod_{N' \in sons(N)} PM(N') & \text{otherwise} \end{cases} \tag{10}$$

For an event type T under the Kleene closure operator, we consider its arrival rate as $2^{R(T)}$ in Equation (10) and multiply by $2^{R(T)} \times w$ Equation (8) since it creates exponential number of events.

4.3 Utility Calculation

DARLING's objective is to maintain a given latency bound L_{max} while trying to maximize the percentage of detected matches (Equation (4)). Therefore, the utility values should reflect the importance of an event to the rate of detected matches. Next, we explain how to create a utility function for each event type; this utility function is used later for on-the-fly utility assignment.

Assume that each event type T has a list of n_T attributes with some global order: $Attrs_T = \{\Lambda_i^T, \forall i \in [0, n_T]\}$, where Λ_i^T represents the i-th attribute of T. Moreover, assume that the probability density function (PDF) of each attribute Λ_i^T is known or can be estimated during the statistics gathering phase, denoted by φ_i^T.

Given condition c, which contains a set of attributes, $Attrs_c = \{\Lambda_i^{T_j}\}$, with known PDFs $\Phi_o^c = \{\varphi_i^{T_j}\}$ (Φ_o^c is the set of original PDFs for c's attributes), we create a new set of PDFs for the condition c:

$$\Phi_n^c = \{\phi_c^{\Lambda_i^{T_j}}, \forall \Lambda_i^{T_j} \in Attrs_c\}$$

Each PDF $\phi_c^{\Lambda_i^{T_j}} \in \Phi_n^c$ takes a value of attribute $\Lambda_i^{T_j} = x$ as its argument and returns the probability of this value passing condition c, given the known PDFs of the other attributes that appear in c:

$$\phi_c^{\Lambda_i^{T_j}}(x) = P(c(\Lambda_i^{T_j} = x) = True \mid \Phi_o^c \setminus \{\varphi_i^{T_j}\})$$

To illustrate the above, assume a condition $c := A.price < B.price$, where $Attrs_c = \{A.price\ (\Lambda_1^A), B.price\ (\Lambda_1^B)\}$ such that $A.price \sim \mathcal{N}(\mu_1^A, \sigma_1^{A^2})$, $B.price \sim \mathcal{N}(\mu_1^B, \sigma_1^{B^2})$, and φ_1^A, φ_1^B are the corresponding PDFs of these normal distributions. Next, we illustrate the computation of $\phi_c^{\Lambda_i^{T_j}}$. For the attribute $A.price$:

$$\begin{aligned} \phi_c^{\Lambda_1^A} &\underset{(1)}{=} P(c(A.price) \mid \Phi_o^c \setminus \{\varphi_1^A\}) \underset{(2)}{=} P(A.price < B.price \mid \varphi_1^B) \\ &\underset{(3)}{=} P(A.price < \varphi_1^B) \underset{(4)}{=} 1 - CDF_{\varphi_1^B}(A.price) \end{aligned}$$

Transition (1) stems from the definition of ϕ, and transitions (2) and (3) are simple assignments. In transition (4) we used the cumulative distribution function (CDF) of $B.price$: $CDF_{\varphi_1^B}$. Eventually, $\phi_c^{\Lambda_1^A}$ gets a value of attribute Λ_1^A ($A.price$) and returns the probability that it will pass the condition c. A similar calculation is performed for $\phi_c^{\Lambda_1^B} = CDF_{\varphi_1^A}(B.price)$. The creation of Φ_n^c is performed periodically, depending on changes in the attributes' PDFs Φ_o^c.

Recall from Section 4.2.3 that $C_{type}(T)$ represents all conditions in the pattern that contain attributes of event type T. The utility function of event e of type T is:

$$U_T(e_T) = \prod_{c_k \in C_{type}(T)} \phi_{c_k}^{\Lambda_i^T}(e_T[i]) \quad \in [0,1] \tag{11}$$

where i is the index of attribute Λ_i^T of event type T that appears in condition c_k, namely $\Lambda_i^T \in Attrs_{c_k}$, and $e_T[i]$ represents the value of this attribute in event e_T. In effect, the utility function $U_T(e_T)$ gets an event e_T of type T and represents the probability that e_T will pass all conditions and be part of a match.

Continuing the above example, assume a pattern $SEQ(A,B)$ $WHERE\ A.price < B.price$, such that $A.price \sim \mathcal{N}(500, 300^2)$ and $B.price \sim \mathcal{N}(400, 200^2)$. Moreover, assume two input events of type A: $a_1(price = 200)$, $a_2(price = 700)$. Then, using $\phi_c^{A.price}$, and $\phi_c^{B.price}$, we get $U_A(a1) = 0.8413$, $U_A(a_2) = 0.066$. Considering the attributes' distributions, a_1 is indeed more likely to appear in matches and is thus assigned a higher utility value.

Calculating the utility values has a negligible effect on the performance of DARLING in overload mode. The utility calculation for event e_T of type T is performed on-the-fly by evaluating Equation (11) using e_T, and depends on $|C_{type}(T)|$. Assume that evaluating each $\phi \in \Phi_n^c$ for arbitrary condition c takes y time units. Then, evaluating $U_T(e_T)$ takes $\alpha_T = |C_{type}(T)| \times y$ time units.

4.4 Auxiliary Data Structures

To enable lightweight load shedding decisions, we introduce new auxiliary data structures that enable DARLING to quickly drop the events of lowest utility.

For each buffer of event type T, DARLING creates ξ_T arrays: $UtilArr_T^0, ..., UtilArr_T^{\xi_T}$. When event e_T of type T with utility $u \in$

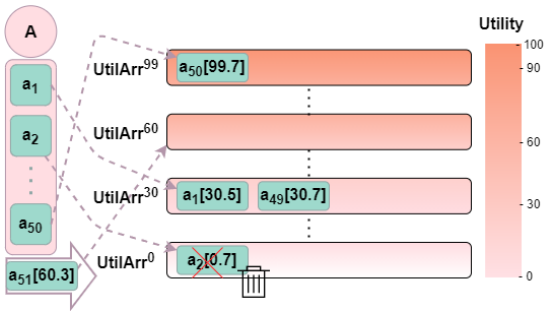


Figure 5: An example of auxiliary data structures for event type A, when the number of arrays is $\xi_A = 100$. Scaled utility values appear in square brackets.

$[0, 1]$ arrives at the system, DARLING scales its utility to $u' = u \times \xi_T \in [0, \xi_T]$. The event is then appended to the buffer of T, and a pointer to e_T is appended to the corresponding array: $UtilArr_T^{\lfloor u' \rfloor}$.

Figure 5 depicts these data structures for event type A. We create $\xi_A = 100$ auxiliary arrays containing pointers for events from the buffer of A. $UtilArr^i$ will contain pointers to events having $\lfloor u \times \xi_T \rfloor = i$. When event a_1 with a utility value of $u = 0.305$ arrives at the system, its utility is scaled to $u' = u \times 100 = 30.5$ and its pointer is appended to the corresponding buffer $UtilArr_A^{\lfloor 30.5 \rfloor}$.

If the global constraint is violated, and the buffer size of event type T is greater than its local constraint N_T, the event with minimum utility is dropped from the buffer. This drop is performed by accessing the minimum-utility array, which is non-empty, and dropping a random event from there. In Figure 5, assume that the global constraint is violated. When event a_{51} arrives at the system and the local constraint $N_A = 50$ is violated, we access the lowest utility buffer $UtilArr_A^0$ and drop a random event from there (a_2). Now, a_{51} can be added to array $UtilArr_A^{\lfloor 60.3 \rfloor} = UtilArr_A^{60}$.

As the number of arrays ξ_T increases, the probability of collisions decreases, i.e., the probability that $\lfloor u_1 \times \xi_T \rfloor = \lfloor u_2 \times \xi_T \rfloor$ for $u_1 \neq u_2$. This means that less events will be in the same array and enable more accurate and granular dropping by utility values. However, the system consumes more memory when it creates more arrays.

The complexity of inserting and dropping events from these data structures involves appending a value to an array, removing a random value from the array, and maintaining the minimum-utility array, which is negligible and does not depend on the array size. The memory used by these data structures for event type T is $(N_T + \xi_T) \times B$ where B is the number of bytes needed for storing a pointer. The overall memory complexity of the data structures is $\sum_{T \in TYPES} (N_T + \xi_T) \times B$ bytes, which is negligible overhead compared to the buffers and the exponential PMs memory usage.

The auxiliary data structures are created upon demand and used only during overload situations. We denote γ_T as the threshold that indicates the need to create the auxiliary data structures for the buffer of event type T. As explained previously, each buffer of event type T has a local constraint N_T on its size. To create the auxiliary data structures, we need to calculate utility values for each event in the buffer and assign the event's pointer to the corresponding array. Recall that α_T is the time needed to calculate the utility value for a single event of type T. Hence, $N_T \times \alpha_T$ is the time it takes to calculate utilities for all events in the buffer of T. During this time, the number of events from the buffer that should have been processed is $\lceil \frac{N_T \times \alpha_T}{Z(T)} \rceil$. Therefore, we start creating the new auxiliary data structures before the overload, when the buffer size reaches the threshold $\gamma_T = N_T - \lceil \frac{N_T \times \alpha_T}{Z(T)} \rceil$.

4.5 Potential Extensions

In this section, we describe the extensions that are natively supported by the design of DARLING and will be addressed in our future work.

Popular evaluation mechanisms. The widely used evaluation mechanisms listed in Section 2.1 are represented by a directed acyclic graph (DAG) and only differ in the restrictions on the allowed graph topology. To adapt DARLING to these mechanisms, the cost model (Equations (5) - (10)) should be extended to accommodate additional graph typologies. It should include proper evaluation of the number of PMs created during processing by the new graph topology (Equations (8)-(10)), the corresponding path in which an event is evaluated (Equations (6)-(7)), and the location of the conditions (Equation (5)). We invite the reader to refer to [33] for more details on the cost model.

Multi-pattern scenario. Multi-pattern CEP is an active research area [34, 43, 57]. Support for multi-patterns will affect both the cost model and the utility calculation in DARLING. In multi-pattern CEP, each pattern can have a different importance and a different latency bound requirement. Extending DARLING to comply with different latency bounds will require creating a separate input buffer (Section 4.2.1) and a respective constraint per latency bound requirement. Events will have to be routed to the input buffers according to the tightest constraint on the latency bound for their type.

The local constraints (Section 4.2.2) on the event types' buffers should capture the importance of the different patterns. This can be achieved by weighting the equation in Section 4.2.2, with the maximum weight related to each event type. That will enable less shedding of events related to more important patterns, with respect to the system's core as defined in Equation (4). As discussed in [34], a multi-pattern evaluation mechanism can be represented as a DAG or a forest of DAGs. Extending the cost model for graphs, as defined above, will enable the precise calculation of $Z(T)$ for multi-pattern.

As explained in Section 4.3, each event has a utility value that reflects its probability of being part of a match for a specific pattern, as defined in Equation (11). For a multi-pattern scenario, the utility of an event belonging to multiple patterns can be computed as a weighted average or as the maximum of its utilities as computed for the different patterns that contain it.

Parallel and distributed CEP. DARLING can be executed in parallel and/or on distributed nodes [45] without any scalability issues. This follows from the core design choice of DARLING of dropping events upon their arrival to the system. The only overhead in deploying DARLING with a distributed or parallelized CEP mechanism would be in the statistics gathering phase and when updating the cost model, where the parallelized or distributed components must synchronize. Nevertheless, this overhead is small since it happens only upon detection of concept drift, and when the system has enough available resources.

5 EXPERIMENTAL EVALUATION

5.1 Experimental Setup

Experimental environment. To simulate a resource-constrained environment, we packed our code in Docker containers. Each container was built from an Alpine3.12 Linux image. Unless stated otherwise, we limited the resources for each container to 1 CPU and 4 GB RAM. The Docker containers ran on an Ubuntu 18.04.5 machine equipped with 377 GB RAM and 80 CPUs, each having 20 cores. Each experiment comprised two containers communicating through a TCP raw socket. The first was a generator container that read the dataset from a file and sent the events through the socket at dynamic rates. The second was a processor container, that contained the core logic, including CEP and load shedding. All algorithms were implemented in Python 3.8.3.

Compared algorithms. We compared DARLING to a baseline algorithm and three state-of-the-art algorithms [48, 49, 58]:

• Random baseline - handles overload situations by randomly dropping partial matches from the system.

• eSPICE [48] - drops events from individual windows by using the event type and its position in the window to calculate the event utility for each window.

• hSPICE [49] - drops events from partial matches inside windows; they use the event type, its position inside the window, and the partial match state to calculate the event utility for each partial match in each window.

• ICDE'20 [58] - combines state-based and input-based techniques for load shedding. State-based shedding is achieved by choosing a "shedding set" with the lowest-utility PMs. After dropping all the PMs of the shedding set, input-based shedding is implemented by continuing to drop all the input events that are part of a PM in the shedding set. PMs are divided into classes of consumption and contribution values, using the event attributes as features. This algorithm is referred to as Zhao et. al. in Section 5.2.

Datasets. We used three real-world datasets in the experiments. The first dataset was taken from historical records of the NASDAQ stock market [1]. The data contains a one-year period of updates to stock prices, covering over 2500 stock identifiers with prices updated on a per-minute basis (409,622,891 events). Each event consists of a stock identifier (used as the event type), a timestamp, trading volume, and prices at open, close, high, and low points.

The second dataset was taken from the DEBS 2013 Grand Challenge [2]. The data originates from wireless sensors embedded in the soccer players' shoes and a ball used during a single match; the data spans the duration of the entire game. The sensors for the players produced data at a frequency of 200Hz, while the sensor in the ball produced data at a frequency of 2000Hz. Each row in the dataset contains a sensor identifier, which is used for the event type, along with timestamp, location coordinates, velocity, and acceleration. This dataset contains 49,576,080 records.

The third dataset contains bus GPS data collected from across Dublin by the Dublin City Council's traffic control [3]; the data spans a one-month period and covers more than 80 line IDs, which were used as event types. This dataset contains 33,097,325 records and each record includes a timestamp and 11 additional attributes updated every few seconds or milliseconds. We augmented the data with the distance of the bus from Dublin city center.

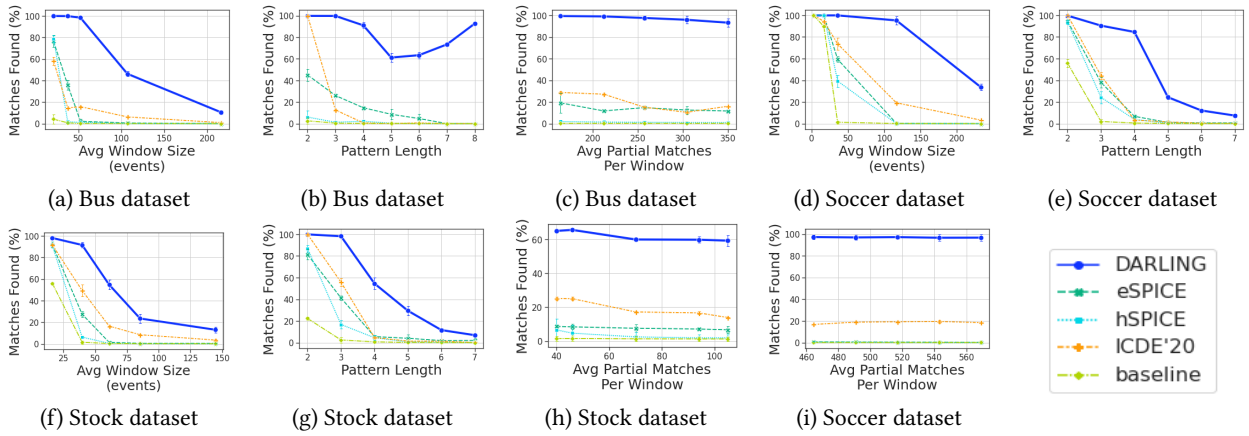
Table 1: Real-world queries for the experiments. Each query operates within a time window of 60 minutes for the stock dataset, 5×10^{10} picoseconds for the soccer dataset, and 10^7 milliseconds for the bus dataset.

Stocks Queries - A	
Q_1^A	**SEQ**(S_1, S_2, ..., S_n) **WHERE** $\forall i \in \{2,n\}$: S_i.volume > S_{i-1}.volume $\times$ C
Q_2^A	**SEQ**(S_1, ..., **KLEENE**(S_j), ..., S_n) **WHERE** $\forall i \in \{2,n\}$: S_i.open > S_{i-1}.open $\times$ C
Q_3^A	**SEQ**(S_1, ...,**NEG**(S_j), ..., S_n) **WHERE** S_j.open < X **AND** $\forall i \in \{2,n\}$: S_i.open > S_{i-1}.open $\times$ C
Soccer Queries - B	
Q_1^B	**SEQ**(GoalB, Ball, GoalA) **WHERE** $dist(GoalB, Ball) < X$ **AND** GoalB.v > GoalA.v $\times$ C
Q_2^B	**SEQ**(Goal, D_1, ...,D_n) **WHERE** $\forall i \in \{1,n\}$: $dist(Goal, D_i) < X$
Q_3^B	**SEQ**(Goal, D_1, ...,**KLEENE**(D_j), ..., D_n) **WHERE** $\forall i \in \{1,n\}$: D_{i-1}.v < D_i.v
Q_4^B	**SEQ**(GoalB, **NEG**(GoalA), Ball) **WHERE** $dist(GoalB, Ball) < X$ **AND** GoalA.v < C
Bus Queries - C	
Q_1^C	**SEQ**(L_1, L_2, ..., L_n) **WHERE** $\forall i \in \{2,n\}$: L_i.dist > L_{i-1}.dist $\times$ C
Q_2^C	**SEQ**(L_1, ...,**KLEENE**(L_j), ..., L_n) **WHERE** $\forall i \in \{2,n\}$: L_i.dist > L_{i-1}.dist $\times$ C
Q_3^C	**SEQ**(L_1, ...,**NEG**(L_j), ..., L_n) **WHERE** L_j.delay > X **AND** $\forall i \in \{2,n\}$: L_i.dist > L_{i-1}.dist $\times$ C

Queries: Table 1 lists the query templates used for each dataset. In the stock dataset, Q_1^A is a query template that represents a sequence of n stock symbols whose trading volume is increasing by a factor of C. Q_2^A represents a sequence of stock symbols with a Kleene closure operator, where the open value increases by a factor of C. Q_3^A represents a sequence of $n-1$ stocks without S_j, or with S_j having $S_j.open < X$. These patterns may imply different relations and correlations between the participating stocks.

In the soccer game queries, Q_1^B represents a sequence for the goalkeeper of team A, a ball event, and the goalkeeper of team B, where the ball is closer to goalkeeper B and the speed of goalkeeper B is higher than the speed of goalkeeper A, hence, team A may score a goal against team B. Q_2^B represents the sequence of a goalkeeper and n defenders from that team; it detects when the distance of all defenders is less than X from the goalkeeper, meaning no possible goal. Q_3^B represents the sequence of a goalkeeper and n defenders whose velocity is increasing, and includes a Kleene closure operator. Q_4^B detects a sequence of goalkeeper B and the ball, where there is either no event of goalkeeper A between them, or one exists but goalkeeper A's velocity is lower than C. These patterns monitor real-time actions such as possible goal (Q_1^B, Q_2^B and Q_4^B) and defense actions (Q_2^B and Q_3^B).

In the bus dataset, queries Q_1^C and Q_2^C detect a sequence of n line IDs with increasing distance from Dublin center. Q_2^C also contains a Kleene closure. Q_3^C detects a sequence of n lines with increased



(a) Bus dataset (b) Bus dataset (c) Bus dataset (d) Soccer dataset (e) Soccer dataset

(f) Stock dataset (g) Stock dataset (h) Stock dataset (i) Soccer dataset

Figure 6: Percentage of matches found (higher is better) under overload rate of 70%, as a function of: average window size ((a),(d),(f)), pattern length ((b),(e),(g)), and average number of partial matches per window ((c),(h),(i)).

distance from the center but does not include L_j, or includes L_j only when its delay is greater than X. These patterns monitor real-time bus trajectories with the Dublin center as a reference point.

The results displayed are the average of multiple independent runs. Statistical significance is shown using error bars, which represent the standard deviation. The maximum standard deviation observed for DARLING was 6% in percentage of detected matches, 12% in latency, and 8% in peak memory consumption.

Implementation details. All algorithms were applied on a left-deep [33] skip-till-any [56] tree evaluation mechanism [37].

Unless stated otherwise, the latency bound L_{max} for each experiment was 1.5 seconds. For all the experiments, we started dropping events after 80% of the input event stream (N_{in}) was full ($\beta = 0.8$).

A warm-up phase preceded the experiments, allowing each algorithm to calculate its statistics and models. During the warm-up phase, no events were dropped. In overload mode (Figures 6, 7, and 8), we sent 70% of the data as load using 4 peaks of 30%, 10%, 20%, and 10%. We create the overload by increasing the predefined arrival rates for these portions of the data by a factor of 1000. The remaining 30% of the data was sent between the load peaks, at the predefined, non-overload arrival rates. We decided to use 4 peaks for the load after we measured the experimental metrics as a function of the number of load peaks and found it to have a limited effect on the performance.

Experimental metrics and parameters. We tested the following performance metrics: (1) *percentage of detected matches* as defined in Equation (4), (2) *latency* as defined in Equation (2), and (3) *peak memory consumption*, the maximum memory required by the system during evaluation. The experimental parameters included: (1) pattern length, the number of event types participating in the pattern; (2) average window size, the average number of events from the pattern that appear in each window; (3) average number of partial matches inside each window; and (4) overload rate. When calculating the percentage of detected matches, we did not consider matches whose latency exceeded 30 minutes. Consequently, the upper limit is 1800 seconds in the latency graphs for overload.

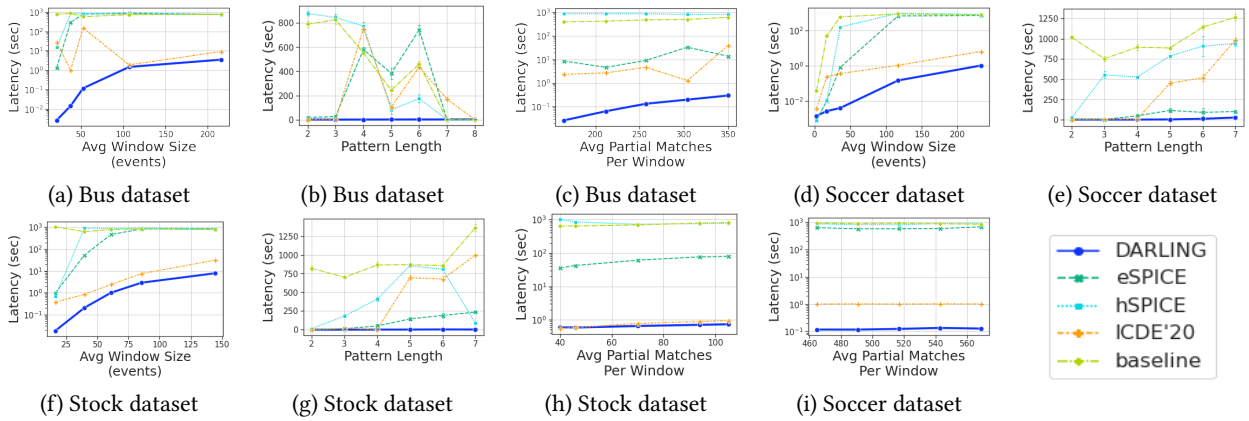
5.2 Experimental Results

In our first experiment, we measured the percentage of matches detected by each algorithm in an overload mode, as a function of the experimental parameters.

For the varying window size parameters ((a), (d), (f), Figure 6), DARLING detected a higher percentage of matches by up to a factor of 4, 12, and 10; these were achieved for the largest window sizes in the stock, bus, and soccer datasets, respectively. Increasing the window size results in more PMs created inside each window and more overlapping between windows.

For each event, eSPICE calculates a different utility value for every window in which it appears, and hSPICE calculates a different utility value for each relevant partial match in these windows. Both eSPICE and hSPICE prevent processing the event within the window or with the PM if the corresponding utility value is below some threshold. The process of calculating utility values for each window or PM is time-consuming and uses up valuable processing time that could otherwise be used for the pattern detection process. In Zhao et. al., increasing the window size leads to more PMs that need to be classified. The exponential number of PMs creates an excess load that should be dropped resulting in more classes of PMs in the shedding set being dropped. Since the input-based technique drops events that appeared in some PM in the shedding set, it potentially drops also important events. The bigger the shedding set, the higher the risk of dropping important input-events. The more overlapping windows and PMs that exist, the more processing time is used for both the pattern detection process and the load shedding decision, resulting in the need for more shedding.

In contrast, DARLING computes a single utility value per event; the processing time for calculating this utility value does not depend on the number of PMs or the number of windows containing the event. The combination of carrying out the load shedding decision upon the arrival of an event according to the global and local constraints on the buffer sizes, and the lightweight calculation of a single utility value per event that properly reflects its importance to the system output, makes DARLING scalable and efficient.



(a) Bus dataset (b) Bus dataset (c) Bus dataset (d) Soccer dataset (e) Soccer dataset

(f) Stock dataset (g) Stock dataset (h) Stock dataset (i) Soccer dataset

Figure 7: Average detection latency (lower is better) under an overload rate of 70%, as a function of: average window size ((a),(d),(f), logarithmic scale), pattern length ((b),(e),(g)), and average number of partial matches per window ((c),(h),(i), logarithmic scale).

We also observed that DARLING outperformed the other algorithms in terms of the average number of partial matches per window ((c),(h),(i), Figure 6) by a factor of 3 to 4 in the stock dataset, 3 to 8 in the bus dataset, and 5 to 6 in the soccer dataset. The displayed results are compatible with the above discussion. A large number of PMs in the system state can considerably impair the system performance by increasing the processing time for each event. Therefore, for a larger amount of PMs more shedding is needed.

The pattern length also affects the rate of detected matches ((b),(e),(g), Figure 6). DARLING achieved a higher percentage of matches by a factor of up to 10, 4836, and 18 for stock, bus, and soccer datasets, respectively. Moreover, for patterns of length 7, it achieved a higher percentage of matches by factor of 3, 4836, and 11 for stock, bus, and soccer datasets, respectively. For the stock and the soccer datasets ((g),(e), Figure 6) we generated longer patterns by adding event types with similar arrival rates to the sequence.

When the pattern gets longer, more PMs are generated hence more shedding is needed. These extended patterns enabled us to examine our load shedding under more rigorous conditions. In the bus dataset ((b), Figure 6), we noticed a rise in the rate of detected matches for pattern lengths greater than 5. This occurred because we appended event types with lower arrival rates to the sequence to create pattern lengths greater than 5; these events do not significantly increase the processing time but do result in fewer matches per window. Since DARLING considers the arrival rate of event types when determining the local buffers constraints (Section 4.2), it does not drop the lower-rate events entirely, and therefore remains able to detect a higher percentage of matches. We further observed that although more PMs were created in longer patterns, DARLING was capable of increasing the percentage of matches due to its data-aware utility calculation and the local constraints.

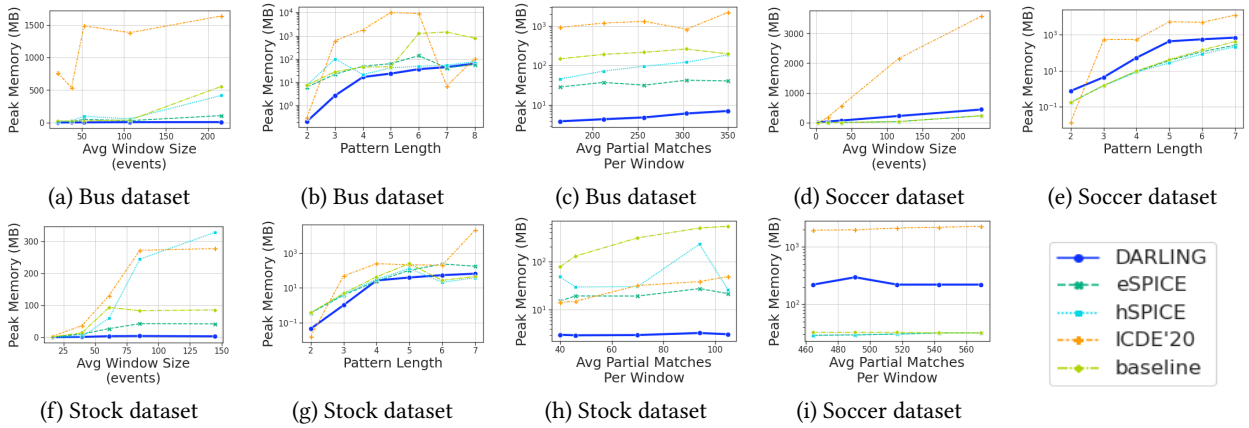
Figure 7 depicts the effect of the experimental parameters on the average detection latency under overload. Recall that we defined a latency bound of $L_{max} = 1.5$ seconds, and that a match is expired if its latency exceeds 30 minutes. While DARLING achieved the lowest detection latency under all experimental parameters and did not exceed L_{max}, the other compared algorithms struggled to maintain the given latency bound.

As the window size increased ((a),(d),(f), Figure 7), the detection latency of state-of-the-art and the baseline algorithms increased. Here, the latency for DARLING was up to 21, 1280, and 87 times lower than that of the lowest-latency algorithm for the stock, bus, and soccer datasets, respectively. Moreover, for the largest window size the detection latency of DARLING was 4, 3, and 7 times lower for the stock, bus and soccer datasets, respectively.

When the pattern length increased ((b),(e),(g), Figure 7), the detection latency of the other compared algorithms increased, and DARLING outperformed the lowest-latency algorithm by up to a factor of 182, 1822, and 46 for the stock, bus, and soccer datasets, respectively. As the pattern gets longer, the number of PMs and the window size increase. Moreover, the more event types that exist in the pattern, the more events being queued, and the higher the detection latency, which depends on the queue size (Section 4.2.1). Since DARLING drops events upon their arrival to the system and performs fast load shedding decisions (Section 4.4), it manages to maintain a fixed size for the queues and preserve the given latency bound. In Figure 7(b) for pattern lengths 7 and 8, the detection latency of eSPICE, hSPICE, and the baseline algorithms is 0, because they did not manage to find any matches. This also applies to Zhao et. al. for a pattern length of 8. Similarly, in Figure 7(g) for a pattern length of 7, the detection latency of hSPICE drops dramatically. This occurs because it found a very small percentage of matches at the beginning of the experiment, when fewer events accumulated in the queues, resulting in smaller latencies.

Under the same number of partial matches per window ((c),(h),(i), Figure 7), the lowest average detection latency for DARLING was 2, 88, and 9 times smaller than the lowest-latency algorithm for the stock, bus, and soccer datasets, respectively.

Figure 8 presents the peak memory consumption comparison. The number of partial matches formed is the most dominant contributor to the memory consumption and is greatly influenced by the load shedding decisions of the algorithm. The memory consumption of the auxiliary data structures in DARLING (Section 4.4) grows linearly with the pattern length and window size, whereas the number of partial matches grows exponentially with both these



(a) Bus dataset (b) Bus dataset (c) Bus dataset (d) Soccer dataset (e) Soccer dataset

(f) Stock dataset (g) Stock dataset (h) Stock dataset (i) Soccer dataset

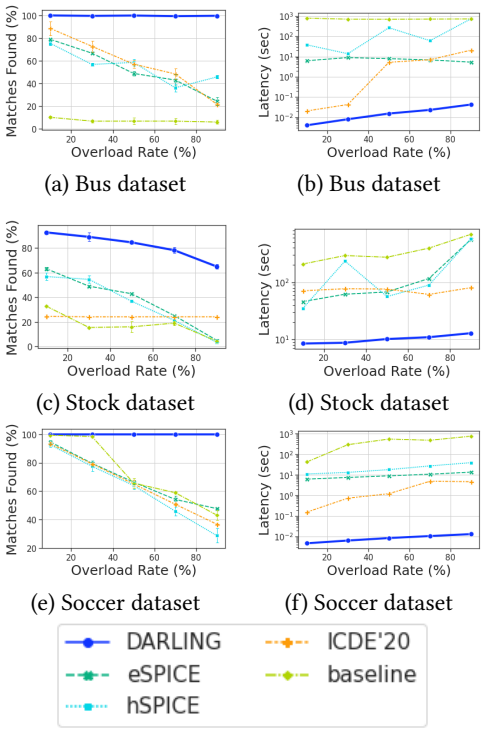
Figure 8: Peak memory consumption (lower is better) under an overload rate of 70%, as a function of: average window size ((a),(d),(f)), pattern length ((b),(e),(g), logarithmic scale), and average number of PMs per window ((c),(h),(i), logarithmic scale).

parameters. Therefore, the memory overhead of these data structures was only 0.03% on average. In most of the experiments, the peak memory usage of DARLING was lower than the peak memory used by the other algorithms. However, in the soccer dataset, which is characterized by higher arrival rates and considerably more partial matches ((d),(e),(i), Figure 8), the memory consumption for DARLING was higher. This can be explained by the fact that the other methods had significantly more shedding than DARLING due to longer processing time, and this decreased their peak memory consumption. Because Zhao et. al. drops PMs after calculating the shedding set (i.e., after the PMs' creation), its peak memory consumption is relatively high.

Figure 9 displays the system performance at a variable rate of overload. In these experiments, the generated load was continuous. DARLING outperformed the other algorithms with respect to the percentage of detected matches ((a),(c),(e), Figure 9) in all datasets. DARLING also demonstrated a detection latency of 7, 123, and 356 times lower in the highest overload rate for the stock, bus, and soccer datasets, respectively. Moreover, DARLING preserves a relatively small latency bound when there is an increase in the load; the other algorithms showed increased latency.

A strong negative correlation between the percentage of found matches to the detection latency can be observed in Figure 9. As the overload rate grows, latencies grow and more events are being dropped in order to maintain the given latency bound, hence the percentage of matches found decreases. In addition, the close-to-zero slope in matches found by DARLING implies that the data-aware utility calculation does a good job of dropping the less important events and keeping the important ones.

The last set of experiments were aimed at measuring the overhead of the load shedding algorithm on the detection latency, when no events are being dropped. The results depicted in Figure 10 show that DARLING achieved the minimal detection latency, which was lower than the latency of the other algorithms by up to 4 orders of magnitude. This can be explained by the utility calculation in DARLING, which does not depend on the experimental parameters. In eSPICE and hSPICE the utility for an event is calculated per window and per PM, and depends on the experimental parameters. It therefore has a high overhead in a setting with overlapping windows and multiple PMs. Moreover, for Zhao et. al., classifying PMs and input events is also time-consuming.

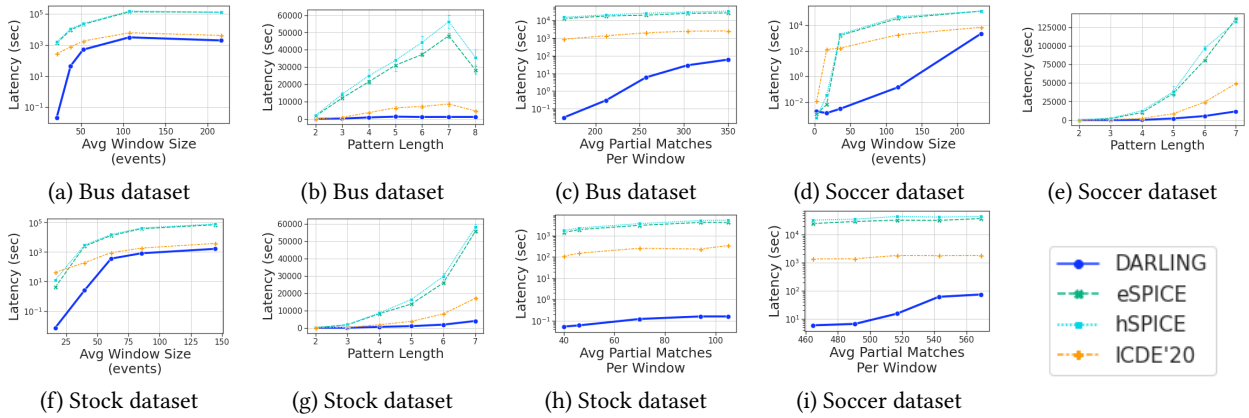


(a) Bus dataset (b) Bus dataset

(c) Stock dataset (d) Stock dataset

(e) Soccer dataset (f) Soccer dataset

Figure 9: Percentage of detected matches ((a),(c),(e), higher is better) and detection latency ((b),(d),(f), logarithmic scale, lower is better), as a function of the overload rate.



(a) Bus dataset (b) Bus dataset (c) Bus dataset (d) Soccer dataset (e) Soccer dataset

(f) Stock dataset (g) Stock dataset (h) Stock dataset (i) Soccer dataset

Figure 10: Detection latency overhead (lower is better) with no dropping of events as a function of: average window size ((a),(d),(f), logarithmic scale), pattern length ((b),(e),(g)), and average number of PMs per window ((c),(h),(i), logarithmic scale).

6 RELATED WORK

In recent years, complex event processing has become an increasingly important research field [17, 18, 22]. The success of earlier data stream management systems [5, 12, 15, 23] led to the development of many general-purpose CEP frameworks [7, 8, 13, 16, 37, 55]. A variety of CEP engines have also been developed [10, 14, 26, 29], alongside multiple CEP libraries [4, 19, 50]. Numerous techniques for optimizing CEP performance have been proposed [16, 19, 20, 28, 40, 55, 56], based on efficient pattern representations [9, 32, 35, 37, 42, 46], sub-expression sharing [9, 19, 34, 36, 43, 57], parallel evaluation [45] and more. These works are orthogonal to DARLING and could be activated by the system prior to load shedding.

Load shedding has gained much attention in the data stream processing domain [30, 31, 38, 39, 41, 44, 51, 53]. Some works [31, 38, 51] use the tuple content to determine its importance. In [51], the authors use random and semantic drops, and in [38] the authors use content-based filters for bounded approximations in answer to aggregate queries. In [31], they use concept-driven load shedding and in [44] high-latency tuples are considered less important. In [41, 53], the authors use reservoir sampling and stratified sampling. In [30] the authors addressed the problem of fair load shedding in federated stream processing systems by balancing the source information content (SIC) metric of different queries. The SIC metric quantifies the amount of source data that was used to create a given query result tuple.

To the best of our knowledge, all methods surveyed in the previous paragraph share common limitations that make them difficult to apply to a CEP setting. Namely, they focus either on aggregations or on two-relational equi-join queries. In aggregate queries, load shedding decisions are made per-tuple, independently of other tuples. However, in CEP, the load shedding decisions must capture the correlations and dependencies created by combining multiple data items. Existing solutions for two-relations equi-joins cannot be trivially adapted to CEP [33], which focuses mainly on complex multi-relational non-equi-joins with models of causality and conceptual hierarchies that contain Kleene closure and negation.

DILoS/ALoMa [39] is the closest in spirit to our work. The authors combined scheduler and load shedder policies to manage stream processing queries of different priority classes. However, while DILoS/ALoMa focuses on "when" and "how much" to shed at the query-class level, DARLING also addresses the "what to shed" question at the event-type level.

Load shedding was recently adapted by CEP [27, 47–49, 58]. In [27], the authors propose integral load shedding, where events of a certain type are either all preserved or all shed, and fractional load shedding, where events are sampled uniformly at random. DARLING generalizes the above approach and makes it possible to shed events based on their importance rather than on frequency alone. In [47], a utility value is assigned to a partial match based on the probability of its completion and its remaining processing time. However, finding partial matches to drop requires iterating over an exponentially large set.

We refer to [48, 49, 58] as our baseline algorithms. eSPICE [48] sheds events from windows, while hSPICE [49] sheds events from PMs. Both methods assume a correlation between events of certain types at certain positions within a window. In [58], the authors propose a hybrid solution that first applies state-based load shedding; it calculates a "shedding set" of lowest-utility PMs to be dropped. The utility of a PM reflects its probability of completion and estimated processing cost. Then, they apply input-based load shedding by dropping all events that appear in some PM in the "shedding set".

7 CONCLUSIONS

In this paper, we studied the problem of load shedding in CEP systems. We formally defined the load shedding problem in CEP and presented DARLING, a novel and scalable load shedding mechanism for CEP. Our extensive experimental evaluation demonstrated significant improvements in performance as compared to the state-of-the-art methods. Our future research efforts will cover the directions presented in Section 4.5.

ACKNOWLEDGMENTS

The research leading to these results was supported by the Israel Science Foundation (grant No.191/18). This research was partially supported by the Technion Hiroshi Fujiwara Cyber Security Research Center and the Israel National Cyber Directorate.

REFERENCES

[1] [n.d.]. http://www.eoddata.com

[2] [n.d.]. https://debs.org/grand-challenges/2013/

[3] [n.d.]. https://data.smartdublin.ie/dataset/dublin-bus-gps-sample-data-from-dublin-city-council-insight-project

[4] [n.d.]. https://www.espertech.com/

[5] Daniel J Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, et al. 2005. The design of the borealis stream processing engine.. In *Cidr*, Vol. 5. 277–289.

[6] Asaf Adi, David Botzer, Gil Nechushtai, and Guy Sharon. 2006. Complex event processing for financial services. In *2006 IEEE Services Computing Workshops*. IEEE, 7–12.

[7] Asaf Adi and Opher Etzion. 2004. Amit-the situation manager. *The VLDB journal* 13, 2 (2004), 177–203.

[8] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. 2008. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 147–160.

[9] Mert Akdere, Uğur Çetintemel, and Nesime Tatbul. 2008. Plan-based complex event detection across distributed sources. *Proceedings of the VLDB Endowment* 1, 1 (2008), 66–77.

[10] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Philippe Selo, Yoonho Park, and Chitra Venkatramani. 2006. SPC: A distributed, scalable platform for data mining. In *Proceedings of the 4th international workshop on Data mining standards, services and platforms*. 27–37.

[11] Leonardo Aniello, Giorgia Lodi, and Roberto Baldoni. 2011. Inter-domain stealthy port scan detection through complex event processing. In *Proceedings of the 13th European Workshop on Dependable Computing*. 67–72.

[12] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. 2016. Stream: The stanford data stream management system. In *Data Stream Management*. Springer, 317–336.

[13] Roger S Barga, Jonathan Goldstein, Mohamed Ali, and Mingsheng Hong. 2006. Consistent streaming through time: A vision for event stream processing. *arXiv preprint cs/0612115* (2006).

[14] Paul C Brown. 2013. *Architecting Complex-Event Processing Solutions with TIBCO®*. Addison-Wesley.

[15] Jianjun Chen, David J DeWitt, Feng Tian, and Yuan Wang. 2000. NiagaraCQ: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 379–390.

[16] Gianpaolo Cugola and Alessandro Margara. 2012. Complex event processing with T-REX. *Journal of Systems and Software* 85, 8 (2012), 1709–1728.

[17] Gianpaolo Cugola and Alessandro Margara. 2012. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)* 44, 3 (2012), 1–62.

[18] Miyuru Dayarathna and Srinath Perera. 2018. Recent advancements in event processing. *ACM Computing Surveys (CSUR)* 51, 2 (2018), 1–36.

[19] Alan Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker White. 2006. Towards expressive publish/subscribe systems. In *International Conference on Extending Database Technology*. Springer, 627–644.

[20] Luping Ding, Songting Chen, Elke A Rundensteiner, Junichi Tatemura, Wang-Pin Hsiung, and K Selcuk Candan. 2008. Runtime semantic query optimization for event stream processing. In *2008 IEEE 24th International Conference on Data Engineering*. IEEE, 676–685.

[21] Opher Etzion and Peter Niblett. 2010. *Event processing in action*. Manning Publications Co.

[22] Ioannis Flouris, Nikos Giatrakos, Antonios Deligiannakis, Minos Garofalakis, Michael Kamp, and Michael Mock. 2017. Issues in complex event processing: Status and prospects in the big data era. *Journal of Systems and Software* 127 (2017), 217–236.

[23] Michael Franklin. 2003. Telegraphcq: continuous dataflow processing for an uncertain world. *CIDR. Asilomar: Morgan Kauf. man Publishers* (2003), 269–280.

[24] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A survey on concept drift adaptation. *ACM computing surveys (CSUR)* 46, 4 (2014), 1–37.

[25] Paulo M Gonçalves Jr, Silas GT de Carvalho Santos, Roberto SM Barros, and Davi CL Vieira. 2014. A comparative study on concept drift detectors. *Expert Systems with Applications* 41, 18 (2014), 8144–8156.

[26] Chetan Gupta, Song Wang, Ismail Ari, Ming Hao, Umeshwar Dayal, Abhay Mehta, Manish Marwah, and Ratnesh Sharma. 2009. Chaos: A data stream analysis architecture for enterprise applications. In *2009 IEEE conference on commerce and enterprise computing*. IEEE, 33–40.

[27] Yeye He, Siddharth Barman, and Jeffrey F Naughton. 2013. On load shedding in complex event processing. *arXiv preprint arXiv:1312.4283* (2013).

[28] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. 2014. A catalog of stream processing optimizations. *ACM Computing Surveys (CSUR)* 46, 4 (2014), 1–34.

[29] Sachini Jayasekara, Srinath Perera, Miyuru Dayarathna, and Sriskandarajah Suhothayan. 2015. Continuous analytics on geospatial data streams with WSO2 complex event processor. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. 277–284.

[30] Evangelia Kalyvianaki, Marco Fiscato, Theodoros Salonidis, and Peter Pietzuch. 2016. Themis: Fairness in federated stream processing under overload. In *Proceedings of the 2016 International Conference on Management of Data*. 541–553.

[31] Nikos R Katsipoulakis, Alexandros Labrinidis, and Panos K Chrysanthis. 2018. Concept-driven load shedding: Reducing size and error of voluminous and variable data streams. In *2018 IEEE International Conference on Big Data (Big Data)*. IEEE, 418–427.

[32] Ilya Kolchinsky and Assaf Schuster. 2018. Efficient Adaptive Detection of Complex Event Patterns. *Proceedings of the VLDB Endowment* 11, 11 (2018).

[33] Ilya Kolchinsky and Assaf Schuster. 2018. Join Query Optimization Techniques for Complex Event Processing Applications. *Proceedings of the VLDB Endowment* 11, 11 (2018).

[34] Ilya Kolchinsky and Assaf Schuster. 2019. Real-time multi-pattern detection over event streams. In *Proceedings of the 2019 International Conference on Management of Data*. 589–606.

[35] Ilya Kolchinsky, Izchak Sharfman, and Assaf Schuster. 2015. Lazy evaluation methods for detecting complex events. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*. 34–45.

[36] Mo Liu, Elke Rundensteiner, Kara Greenfield, Chetan Gupta, Song Wang, Ismail Ari, and Abhay Mehta. 2011. E-cube: multi-dimensional event sequence analysis using hierarchical pattern query sharing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 889–900.

[37] Yuan Mei and Samuel Madden. 2009. Zstream: a cost-based query processor for adaptively detecting composite events. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. 193–206.

[38] Chris Olston, Jing Jiang, and Jennifer Widom. 2003. Adaptive filters for continuous queries over distributed data streams. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 563–574.

[39] Thao N Pham, Panos K Chrysanthis, and Alexandros Labrinidis. 2016. Avoiding class warfare: managing continuous queries with differentiated classes of service. *The VLDB Journal* 25, 2 (2016), 197–221.

[40] Olga Poppe, Chuan Lei, Salah Ahmed, and Elke A Rundensteiner. 2017. Complete event trend detection in high-rate event streams. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 109–124.

[41] Do Le Quoc, Ruichuan Chen, Pramod Bhatotia, Christof Fetzer, Volker Hilt, and Thorsten Strufe. 2017. Streamapprox: Approximate computing for stream analytics. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*. 185–197.

[42] Ella Rabinovich, Opher Etzion, and Avigdor Gal. 2011. Pattern rewriting framework for event processing optimization. In *Proceedings of the 5th ACM international conference on Distributed event-based system*. 101–112.

[43] Medhabi Ray, Chuan Lei, and Elke A Rundensteiner. 2016. Scalable pattern sharing on event streams. In *Proceedings of the 2016 international conference on management of data*. 495–510.

[44] Nicoló Rivetti, Yann Busnel, and Leonardo Querzoni. 2020. Load-aware shedding in stream processing systems. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XLVI*. Springer, 121–153.

[45] Henriette Röger and Ruben Mayer. 2019. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Computing Surveys (CSUR)* 52, 2 (2019), 1–37.

[46] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. 2009. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. 1–12.

[47] Ahmad Slo, Sukanya Bhowmik, Albert Flaig, and Kurt Rothermel. 2019. pSPICE: Partial Match Shedding for Complex Event Processing. In *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 372–382.

[48] Ahmad Slo, Sukanya Bhowmik, and Kurt Rothermel. 2019. espice: Probabilistic load shedding from input event streams in complex event processing. In *Proceedings of the 20th International Middleware Conference*. 215–227.

[49] Ahmad Slo, Sukanya Bhowmik, and Kurt Rothermel. 2020. hSPICE: state-aware event shedding in complex event processing. In *Proceedings of the 14th ACM International Conference on Distributed and Event-based Systems*. 109–120.

[50] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. 2011. Siddhi: A second look at complex event processing architectures. In *Proceedings of the 2011 ACM workshop on Gateway computing environments*. 43–50.

[51] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. 2003. Load shedding in a data stream manager. In *Proceedings 2003 vldb conference*. Elsevier, 309–320.

[52] Kia Teymourian, Malte Rohde, and Adrian Paschke. 2012. Knowledge-based processing of complex stock market events. In *Proceedings of the 15th International Conference on Extending Database Technology*. 594–597.

[53] Wee Hyong Tok, Stéphane Bressan, and Mong-Li Lee. 2008. A stratified approach to progressive approximate joins. In *Proceedings of the 11th international*

conference on Extending database technology: Advances in database technology. 582–593.

[54] Alexander Widder, Rainer von Ammon, Gerit Hagemann, and Dirk Schönfeld. 2009. An Approach for Automatic Fraud Detection in the Insurance Domain.. In *AAAI Spring Symposium: Intelligent Event Processing*. 98–100.

[55] Eugene Wu, Yanlei Diao, and Shariq Rizvi. 2006. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 407–418.

[56] Haopeng Zhang, Yanlei Diao, and Neil Immerman. 2014. On complexity and optimization of expensive queries in complex event processing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 217–228.

[57] Shuhao Zhang, Hoang Tam Vo, Daniel Dahlmeier, and Bingsheng He. 2017. Multi-query optimization for complex event processing in SAP ESP. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 1213–1224.

[58] Bo Zhao, Nguyen Quoc Viet Hung, and Matthias Weidlich. 2020. Load Shedding for Complex Event Processing: Input-based and State-based Techniques. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1093–1104.