Meeting Predictable Buffer Limits in the Parallel Execution of Event Processing Operators

Ruben Mayer, Boris Koldehofe and Kurt Rothermel
Institute for Parallel and Distributed Systems
University of Stuttgart
Stuttgart, Germany
Email: firstname.lastname@ipvs.uni-stuttgart.de

***Abstract*—Complex Event Processing (CEP) systems enable applications to react to live-situations by detecting event patterns (complex events) in data streams. With the increasing number of data sources and the increasing volume at which data is produced, parallelization of event detection is becoming of tremendous importance to limit the time events need to be buffered before they actually can be processed by an event detector—named event processing operator. In this paper, we propose a pattern-sensitive partitioning model for data streams that is capable of achieving a high degree of parallelism for event patterns which formerly could only be consistently detected in a sequential manner or at a low parallelization degree. Moreover, we propose methods to dynamically adapt the parallelization degree to limit the buffering imposed on event detection in the presence of dynamic changes to the workload. Extensive evaluations of the system behavior show that the proposed partitioning model allows for a high degree of parallelism and that the proposed adaptation methods are able to meet the buffering level for event detection under high and dynamic workloads.**

***Keywords*—Complex Event Processing, Stream Processing, Data Parallelization, Self-Adaptation, Quality of Service**

I. Introduction

For modern IT systems, the ability to *timely* react to situations of interest occurring in the real world has become a fundamental requirement. A growing number of spatially distributed sensors allow for monitoring the physical world and enable new situation-aware applications, e.g., in algorithmic trading [1], traffic monitoring [2]–[5] and the management of smart energy grids [6]. Distributed *Complex Event Processing (CEP)* systems [7], [8] provide the integration and analysis of data streams from such sensors, correlating the low-level information in the sensor streams to complex events that correspond to situations of interest. Along this process, distributed *CEP operators* detect event patterns on incoming event streams and emit new events to neighboring operators or to event consumers. In many cases, timely event detection is crucial, as events that are detected late reduce the benefit the application can gain from the CEP system.

Delays in an event processing system have several causes, such as delays imposed by communication, processing, and buffering of events. In a managed environment like a data center it is in many cases easy to find acceptable bounds for communication and processing latencies that can be met with high probability. For instance, for an operator applied in a traffic monitoring scenario (cf. Section VI-B) we have measured in 99 % of the cases a maximal delay of 1.25 seconds for processing one event. However, understanding communication and processing delays is not enough when the operator is overloaded, i.e., the arrival rate of events exceeds the achievable processing rate. In this case a high amount of events need to be buffered before being processed, which can cause an unacceptable latency in event detection. For instance, for the former traffic monitoring operator, several thousand buffered events induce a buffering delay in the magnitude of minutes or even hours. Thus, in order to timely detect events, it is important to be able to establish predictable *buffering limits* for the operator.

This is highly challenging, since many situations of interest encompass a high and fluctuating number of monitored entities or events. For instance, in the New York Stock Exchange (NYSE) in average 215,162 quotes/s and 28,375 trades/s occur, with peak rates of 308,705 quotes/s and 49,570 trades/s [9]. Even higher fluctuations can be observed in traffic monitoring: Official traffic statistics from the California Department of Transportation[1] show that in one single hour ("rush hour") up to 25 % of the total daily traffic volume can occur on streets. Those numbers show that when analyzing stock markets or traffic situations, operators have to be highly scalable and adaptive in order to keep a buffering limit.

To reach a processing rate that is high enough to cope with the workload, the efficient parallel execution of an operator on dozens of processing entities is necessary. Amongst all other operator parallelization approaches, *data parallelization* frameworks [10]–[17] that apply stream partitioning [18] are the most potent ones in achieving a high degree of parallelism. Such frameworks are capable of partitioning the incoming event stream of an operator, process the partitions in parallel on several operator instances, and then merge the events produced by the instances to an outgoing event stream. However, state-of-the-art frameworks do not provide methods for consistent stream partitioning for many operator types, for example aperiodic, period and sequence operators as defined in the operator specification language Snoop [19]. This can lead to *false-negatives* and *false-positives*. Furthermore, when facing fluctuating workloads, the parallelization degree must be continuously adapted in order to keep a buffering limit. However, since fluctuations of the workload may have a delayed effect on the imposed processing efforts in event detection, state-of-the-art approaches [20], [21] that only react to changes in the utilization of resources will not be able to ensure a predictable buffering limit. As also confirmed later by our

[1] http://traffic-counts.dot.ca.gov/

evaluation results, those reactive approaches impose to exceed the acceptable buffering limit most of the time by a factor of more than 1000.

In this paper, we make an important step towards CEP systems that timely detect events, by ensuring that each operator in an operator network produces consistent results and enforces predictable buffering limits. Our contributions are threefold: First, we propose a novel *pattern-sensitive* stream partitioning model. The partitioning model allows to consistently parallelize a wide class of CEP operators and ensures a high degree of parallelism. Second, we propose methods to model the workload and dynamically adapt the parallelization degree utilizing *Queuing Theory (QT)*, so that a buffering limit of each operator can be met with high probability. We combine the proposed models and methods, building an adaptive data parallelization middleware. Our evaluation shows that the proposed stream partitioning methods can achieve a high throughput of up to 380,000 events per second, even on commodity hardware. Moreover, we show in the context of a traffic monitoring scenario that under heavily fluctuating workloads the adaptation methods enforce a stable parallelization degree and this way ensure that the buffering limit is met and only little over-provisioning in terms of resources is required.

The paper is structured as follows. Section II provides an overview of our data parallelization middleware. Section III describes the problem that we tackle with the system and formalizes the guarantees about the buffering limit that we provide. The novel stream partitioning model is described in Section IV. In Section V, the problem of adapting the operator parallelization degree is tackled. Finally, in Section VI, we provide an extensive evaluation of the system, showing the performance and efficiency in the context of exemplary scenarios in the field of traffic monitoring. Related work is discussed in Section VII, before we conclude the paper and give an outlook on future work in Section VIII.

II. System Overview

A. Distributed CEP System

A distributed CEP system consists of an operator graph, incorporating event sources, operators, and event consumers that are interconnected by event streams. An *event* e contains a tuple of attribute-value pairs (the *payload* of the event), a unique *sequence number* within its stream and a time stamp. Events of different streams inherently possess a well-defined global order by their time stamps, sequence numbers and source. Each operator processes events in-order on its incoming streams and emits events on its outgoing streams to its successors in the graph. The set of events in all incoming or outgoing streams of an operator ω is denoted I_ω or O_ω respectively.

B. Data Parallelization Architecture

In our data parallelization middleware, each operator in the operator graph is executed in a *data parallelization framework* (cf. Figure 1): A splitter is partitioning I_ω into independently processable partitions, a number of operator instances process in parallel the assigned partitions and produce new events, and a merger orders the events emitted by the operator instances into a deterministic sequence O_ω. In doing so, the framework

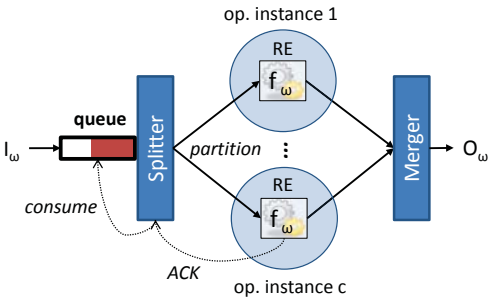


Fig. 1. Data parallelization framework.

dynamically adapts the number of operator instances, labeled the *parallelization degree*.

1) Splitter: Events arriving on the incoming streams are stored in-order in a queue. The splitter assigns the events according to a *partitioning model* to different operator instances. The instances process the events of the assigned partitions, and acknowledge them as soon as they are processed. When an event has been acknowledged by all operator instances which it has been assigned to, the event is consumed, i.e., discarded from the splitter queue. This way, the queue grows or shrinks depending on the ratio of acknowledged events to newly arriving events. We assume that the complexity in splitting the event streams is by magnitudes lower than the complexity in the event detection, so that the splitter does not become the bottleneck in the operator. This assumption is proven to be realistic by our evaluation results which show a high throughput in the splitter of up to 380,000 events per second, even on commodity hardware (cf. Section VI-D).

2) Operator Instances: The execution of an operator instance is controlled by a runtime environment (RE). This comprises for the main part the management of partitions and communication with the splitter. The RE receives information about the bounds of the assigned partitions and the corresponding events, and manages the operator execution so that the assigned partitions are processed. When an event is processed, the RE acknowledges it at the splitter.

3) Merger: The merger ensures that an ordering between all produced events is established, if such an ordering is required at subsequent operators or event consumers. Furthermore, the merger assigns consecutive sequence numbers to the ordered events.

C. System Model

The middleware is deployed on an infrastructure that consists of a number of computing nodes that are considered failure-free and provide a homogeneous computing capability in terms of CPU and memory[2]. The number of hosts used by the data parallelization middleware is flexible. A sufficient number of hosts is available, so that new hosts can be allocated for the deployment of operator instances as well as deallocated

[2]To work with heterogeneous nodes, the methods we developed in this paper can be extended to take into account individual computational capabilities of nodes by utilizing operator profiles for all available node types (cf. Section V-B). However, homogeneous capabilities are a common case in virtualized computing environments, so that the assumption is practical to focus on the main challenges that are tackled in this paper.

when they are not used any more. The allocation of a new host and deployment of an operator instance is assumed to take TH *(Time Horizon)* time units from the allocation request until the instance is available. The hosts are connected by communication links which guarantee eventual in-order delivery of data.

III. Problem Description

For each operator ω, the following guarantees must be provided: Given a correct prediction of the *probabilistic distribution* of inter-arrival times of events TH time units in the future, the parallelization degree is adapted so that a user-defined buffering limit BL_ω of buffered events in the queue of the splitter will be kept with a user-defined probability $P_{required}$. For example, if the buffer limit is 100 with $P_{required}$ = 95 %, then the filling level of the queue at the splitter will be below 100 events with a probability of 95 % when checked at an arbitrary point in time.

This requires solutions to two subproblems: Consistent stream partitioning (Problem 1) and the adaptation of the parallelization degree to fluctuating workloads (Problem 2).

a) Problem 1: Consistent stream partitioning: Many applications utilizing CEP systems require that detected events capture the status of the monitored entities in the real world in a consistent way, so that no events of interest to the consumer are disregarded *(false-negative)* as well as no events that have not occurred are signaled (*false-positive*) [22]. It is therefore important for the splitter to find *consistent* partitioning points in the incoming event streams, so that operator instances produce exactly those events that would be produced in a sequential execution of the operator. For instance, consider a CEP operator applied in traffic monitoring that controls a no-passing zone for vehicles between two control points. A false-positive detection of a truck passing another road user would cause an unjustified ticket, while a false-negative detection would leave the transgressor unpunished.

To increase the processing rate of an operator, stream partitioning in the splitter must happen independently of processing the partitions in the operator instances. Furthermore, the operator instances must process the assigned partitions while sharing only minimal state amongst each other and needing only minimal synchronization effort. Consistent stream partitioning must be supported for a wide range of operators. In doing so, the splitter should work with a simple model that only requires minimal knowledge about the operator logic that is needed in order to partition the incoming streams. Executing the operator in the parallelization framework should be possible without interference with the operator logic.

b) Problem 2: Adaptation of the Operator Parallelization Degree: It would be possible to determine the optimal number of instances with a simple mathematical equation if the inter-arrival times of events at the splitter and the processing rates of events at operator instances were fixed and well-known. However, often the workload and also the processing rates fluctuate. For instance, in the traffic monitoring scenario, between night time and rush hour a huge difference in the average rate of vehicles on the road can be observed, and also at a constant average rate, the number of vehicles per time unit is bursty. To keep the parallelization degree minimal instead of always providing for a worst case workload, it must be continuously adapted. The adaptation method has to take into account TH, the time needed to deploy a new instance.

```
1: int nextStart = 0, slideTime, windowSize

2: bool Ps (Event e) begin
3:   if e.timestamp ≥ nextStart then
4:     nextStart = e.timestamp + slideTime - (e.timestamp % slideTime)
5:     return TRUE
6:   else
7:     return FALSE
8:   end if
9: end function

10: bool Pc (Event e, Selection s) begin
11:   if e.timestamp > (s.startTime + windowSize) then
12:     return TRUE
13:   else
14:     return FALSE
15:   end if
16: end function
```

Fig. 2. Predicates for the time sliding window operator.

IV. Stream Partitioning

To find partitions in the event streams that yield consistent processing results in the operator instances without requiring any adaptations of the operator logic, we first analyze how an operator works on the event streams and describe it in a general *event processing model*. Then we develop a *partitioning model* that is capable to partition the streams in such a way that the operator instances produce consistent streams according to the event processing model.

A. Event Processing Model

Generally, a property of CEP operators is that they perform a sequence of independent *correlation steps* on the events of the incoming streams [22]. In each step, a finite, non-empty set of events denoted *selection* σ is correlated and a finite tuple of events is produced and emitted to the merger. This can be represented by a mapping of tuples of incoming events $\mathbb{E}_{in}$ to tuples of outgoing events $\mathbb{E}_{out}$, which is denoted by the *correlation function* $f_\omega : \mathbb{E}_{in} \mapsto \mathbb{E}_{out}$. When correlating events of a selection, the instance executes *computations* on the events in σ. Such computations can be as simple as applying a filter rule or as complex as running a face recognition algorithm on a set of video frames that are contained in the event payload. As the correlation function is a mapping, between two correlation steps no computational state is maintained.

B. Partitioning Model

The idea of our partitioning model is to split the incoming event stream *by selections* which by definition contain all events needed to detect the comprised event patterns, so that we refer to the model as *pattern-sensitive stream partitioning*. That means that each partition must comprise one or more complete selections, i.e., all events that are part of the selection(s). As no computational state is maintained between the processing of different selections, executing two different correlation steps on two different operator instances is possible without any need to share state between the two operator instances.

TABLE I. SUPPORT OF CONSISTENT PARTITIONING. 'X' DENOTES SUPPORTED, '–' DENOTES NOT SUPPORTED.

	Ti	Tu	∨	;	∧	A	P
Pattern-sensitive	x	x	x	x	x	x	x
Run-based	–	x	x	–	x	–	–
Key-based	–	–	x	–	x	–	–

To ensure that a selection is completely comprised in a partition, all events between the first and the last event of the selection must be part of that partition. Thus, to partition I_ω, the points where selections start and end must be determined. For each event in I_ω, one or more out of three possible conditions are true:

(i) The event triggers that a new selection is opened.
(ii) The event is part of open selections.
(iii) The event triggers that one or more open selections are closed.

To evaluate which condition is true, the splitter offers an interface that can be programmed according to the operator functionality and that provides the ability to store variables that capture internal state, e.g., about window start times. The interface comprises two predicates: $P_s : e \rightarrow \text{BOOL}$ and $P_c : (\sigma_{open}, e) \rightarrow \text{BOOL}$. For each incoming event e, P_s is evaluated to determine whether e starts a selection, and P_c is evaluated with each open selection σ to determine whether e closes σ. Depending on the order of the evaluations of P_s and P_c on a newly arrived event, different semantics can be realized with respect to whether the start and close events are part of the selection or not.

In Figure 2, we showcase the predicates for a time-based sliding window operator [23]. Selections are spanned over all events within a time window that moves by a sliding parameter for each new selection.

C. Runtime Environment

A runtime environment (RE) manages the execution of an operator instance. It receives the partitions assigned by the splitter and ensures that exactly those correlation steps are executed that correspond to the assigned partitions. In particular, when assigned partitions have an overlap in the set of comprised events, the RE prevents the operator instance from detecting patterns of partitions that have not been assigned to it, but enforces an isolation between the processing of the different partitions. This is done in the following way: When the operator instance processes an event that potentially starts a new correlation step, a function in the RE `isAssigned(startevent)` is called that signals whether it is a start event of an assigned partition or not. Further, when an event has been processed in all assigned partitions it is comprised in, the RE sends an acknowledgment to the splitter.

D. Expressiveness

To analyze the operators for which consistent stream partitioning is supported, we consider different categories of event processing operators from the literature. The following categories are considered: Sliding window operators (time-based (Ti) and tuple-based (Tu)) [23], disjunction (∨), sequence (;), conjunction (∧), aperiodic (A) and periodic (P) operators (all [19]) utilizing an unrestricted consumption mode [24], i.e., events can be used in several partitions without restrictions. Table I compares our pattern-sensitive stream partitioning, the run-based approach proposed in [1] where event streams are partitioned into batches of a fixed size, and the key-based approaches like [11]–[14], [16], [17] where event streams are partitioned by a key contained in the events. Run-based partitioning cannot work consistently with operators for which the maximal amount of events comprised in a partition depends on the occurring events and thus is unknown before run-time, which is the case for (Ti), (;), (A) and (P) operators. Key-based partitioning is even less expressive, not allowing for the consistent partitioning of event streams based on any other information than keys contained in every single event, so that the context of an event cannot be considered as it would be needed in (Ti), (Tu), (;), (A) and (P) operators. Pattern-sensitive stream partitioning does not suffer from those limitations. For (Ti) (cf. Figure 2), (Tu), (;), (A) and (P), the internal state in the splitter can store the necessary information about the selection start event. The predicates for (∨) and (∧) are based on the simple comparison of event types.

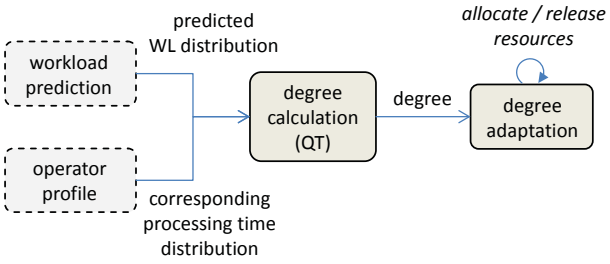


Fig. 3. Workflow of the adaptation of the parallelization degree.

V. ADAPTING THE PARALLELIZATION DEGREE

In this section, we describe how to find and automatically adapt the parallelization degree at changing workloads. We aim to always deploy the optimal parallelization degree, which is the minimal degree that allows the operator to keep the assigned buffer limit with the required probability. To achieve this goal, we rely on *Queuing Theory (QT)* [25] to deduct a stationary distribution of the splitter queue length for a given parallelization degree. According to QT, we model the workload and processing time of an operator by probabilistic arrival and service processes of events. According to the arrival process, the events in I_ω determine the *workload* of ω. They are streamed from the predecessors of ω, resulting in a process of inter-arrival times of events. According to the service process, the *processing time* of an instance of ω is described as the time it takes to process an event. This way, it is possible to dimension the operator solely based on the observed arrival and service processes without interference to the internal operator logic.

This allows us to generate the following workflow in our approach (cf. Fig. 3): Models of the predicted future workload and of the corresponding processing times are used for continuously calculating the optimal parallelization degree with QT methods. The calculated degree is established by an

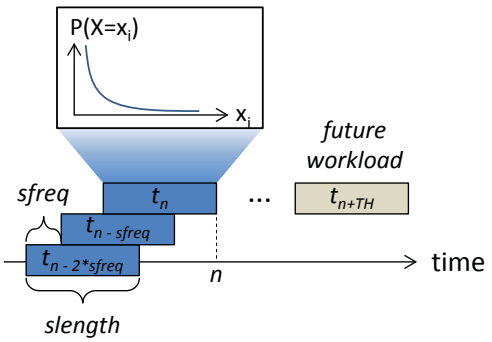


Fig. 4. Workload of an operator in time slices. The highlighted time slice shows an exponential distribution of inter-arrival times x.

```
1: function calculate_degree (WL) begin
2:   c = current_degree                          ▷ parallelization degree
3:   while true do
4:     P = ∑_{n=1}^{BL_ω} P(Q(t) = n)            ▷ applying QT formulas
5:     if P < P_required then
6:       c = c + 1
7:     else
8:       if P ≥ P_required AND last_P < P_required then
9:         return c
10:      else
11:        c = c − 1
12:      end if
13:    end if
14:    last_P = P
15:  end while
16: end function
```

Fig. 5. Algorithm calculating the optimal parallelization degree.

adaptation algorithm that allocates and deallocates operator instances.

A. Workload Monitoring and Prediction

A typical workload is characterized by three parts [26]: Seasonal behavior, trends and noise. To account for short term fluctuations around an average value, i.e., noise, as well as medium or long term changes, i.e., seasonal behavior and trends, we divide the time scale into *time slices* (cf. Figure 4) and denote by t_n the time slice that ends at a point in time n. To this end, we utilize a sliding time window with a length of $slength$ time units that for each new time slice moves by $sfreq$ time units. In each time slice, the inter-arrival time of arriving events follows a *probabilistic distribution*. Parameters of the distribution, for example, the mean value or the variance, can change over subsequent time slices. In Figure 4, the distribution of inter-arrival times X in a time slice t_n is depicted, reflecting an exponential distribution.

The splitter logs the inter-arrival times of events. In order to automatically match the logged data set of a time slice to a probabilistic distribution of inter-arrival times, we have developed a workload classification algorithm that comprises three basic steps [27]: The algorithm iterates over a set of possible distributions (exponential, deterministic, normal, uniform, and Pareto distributions), the parameters of the selected distribution are estimated, and the goodness of the match with respect to the log is determined. The distribution that fits best to the logged data is chosen.

To estimate the parameters of the selected distribution, the algorithm chooses between two approaches, depending on the selected distribution. When the parameters can be described as a function of the moments of the distribution function, the method of matching the *distribution moments* [27] is applied. This method is chosen when no higher moments are needed, i.e., for the exponential, deterministic, normal or uniform distribution. Otherwise, the *maximum likelihood method* [27] is applied. With this method, the parameters are calculated such that they would, when sampled from the selected distribution, lead to the given logged data with the highest probability.

To check the goodness of the selected distribution, statistical tests are applied [27], [28]. An important kind of goodness-of-fit test are tests that utilize statistics based on the *empirical distribution function (EDF)* [28]. The downside of this approach is that the necessary statistic tables are only available for standard distributions like the normal or the exponential distribution. In other more sophisticated cases, the algorithm utilizes the χ^2 *test for homogeneity*, where random samples from the selected distribution are created and checked against the monitored data [29]. This has the advantage that the hypothetic distribution is not directly included in the analysis, but only indirectly through the samples.

In order to predict the future workload TH time units ahead, we utilize methods from time series analysis on the workload monitored in the latest time slices [26]. Depending on the distribution that has been monitored, a different number of parameters will be predicted. For instance, when considering an exponential distribution, only the mean value is relevant, while in a Normal distribution also the variance is subject to prediction.

B. Operator Profiling

The processing time of an instance often depends on the workload: For example, for an operator aggregating values in a time-based window, higher event rates mean that the processed windows contain more events, so that the processing time in the window increases. To account for that, the operator is profiled before run-time for different workloads on the target system. To create an *operator profile*, an operator instance is instrumented, and for different workload distributions the distribution of processing times is measured.

C. Degree Calculation

Depending on the predicted workload at time slice t_{n+TH} and the operator profile, the algorithm listed in Figure 5 calculates the parallelization degree that is necessary in order to keep BL_ω in that time slice. The algorithm utilizes methods from QT to compute the probability to keep BL_ω for a given parallelization degree. Let Q = {Q(t) : t $\geq$ 0} be the random process of the queue length Q(t) at time t. Depending on the workload distribution, the corresponding processing time distribution and the number of operator instances, the distribution of Q(t) can settle down for $t \to \infty$ to a stationary distribution, so that the queue length probability P(Q(t) $\leq BL_\omega$), i.e., the probability that there are not more than BL_ω buffered in the queue, can be calculated. Formulas to calculate P(Q(t) $\leq BL_\omega$)

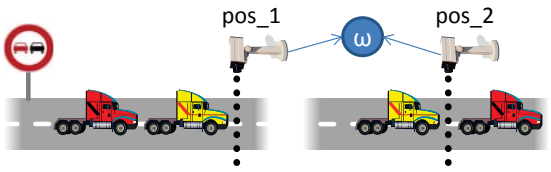


Fig. 6. Surveillance of a no-passing zone utilizing two sensors.

are available for M/M/c [30] and M/D/c [31] queuing systems[3]. The algorithm uses the available mathematical formulas to compute the queue length probabilities P(Q(t) $\leq BL_\omega$) (line 4). This way, it is analyzed whether for a given parallelization degree c, the queue length keeps the buffering level limit with a probability that is higher than $P_{required}$ (line 5). c is changed accordingly, until the minimal c is found for which the stationary queue length distribution yields $P \geq P_{required}$ (line 8), which is returned as the optimal parallelization degree.

However, for many distributions Q(t) does not settle down to a stationary distribution, and in practice not all predicted workloads and profiled service time distributions are exponentially or deterministically distributed. In that case, they are *approximated* by an exponential or deterministic distribution such that the *cumulative distribution function* (cdf) of inter-arrival times is higher than the cdf of the actual distribution. That way, the approximation yields smaller inter-arrival times. When approximating the processing time distribution, it is done in the opposite way: The cdf of the approximation must be smaller to yield larger processing times. Note that even in the presence of such approximations it is still necessary to have a precise workload and service time classification in order to make sure the approximation yields smaller inter-arrival times and higher processing times than the approximated distribution.

D. Adaptation

Adaptation of the *parallelization degree*. The adaptation of the parallelization degree is initiated every *sfreq* time units. The new optimal parallelization degree is calculated with the algorithm from Section V-C based on the predicted workload in TH time units. If there is a difference dif between the new degree and the last updated parallelization, the adaption of the degree in TH time units is prepared. There are two cases: (i) $dif < 0$ and (ii) $dif > 0$. In case (i), the cancellation of $|dif|$ instances in TH time units is registered. At the time of cancellation, the operator instances are unregistered in the splitter, so that they do not receive new selections. The instances finish processing the assigned partitions and then shut down. In case (ii), in order to reduce the number of deployment operations, it is first checked how many canceled instances are not shut down yet and can be reused. Those n instances are registered in the splitter. The rest of the $dif - n$ instances are newly deployed and registered in the splitter as soon as they are live.

Analysis of *slength* and *sfreq*. $slength$ must be long enough to capture a sufficient number of events that capture the workload distribution with high confidence, but also short enough so that workload trends can be detected in time. For instance, considering the mean value in an exponential distribution, in order to reach a confidence interval of 95 % that does not exceed $\delta = \pm 5\%$ of the sample mean, following an approximation in [32] $slength$ is chosen such that the number of measurements n is 1600, while for $\delta = \pm 10\%$ a value of $n = 400$ is sufficient. In setting $sfreq$, it must be small enough so that changes in the workload are detected timely, but the higher it is the less computational overhead is caused. Both $slength$ and $sfreq$ depend on the queue length requirements, the operator and the workload, and need to be determined experimentally (cf. Section VI-B4).

[3]We follow Kendall's notation, A/B/c, A = workload distribution, B = processing time distribution, c = number of services. M = exponential distribution / Markovian process, D = deterministic distribution / constant.

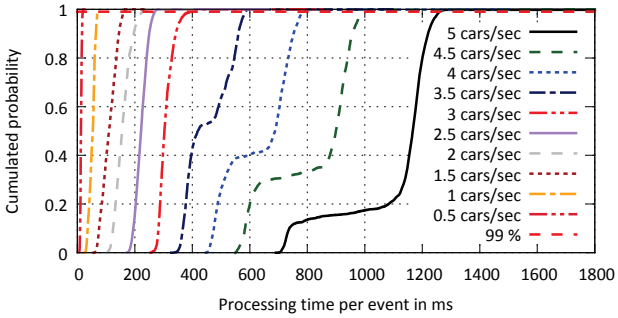


Fig. 7. Operator profiles.

VI. Evaluation

In this section, we evaluate the proposed method to adapt the operator parallelization degree by analyzing the queue lengths at different workloads. We compare the QT-based approach to an approach based on CPU measurements, building on a traffic monitoring scenario. Furthermore, we analyze the effects of approximating given workload and service time distributions. Finally, we evaluate the performance of the splitter in partitioning the incoming event streams.

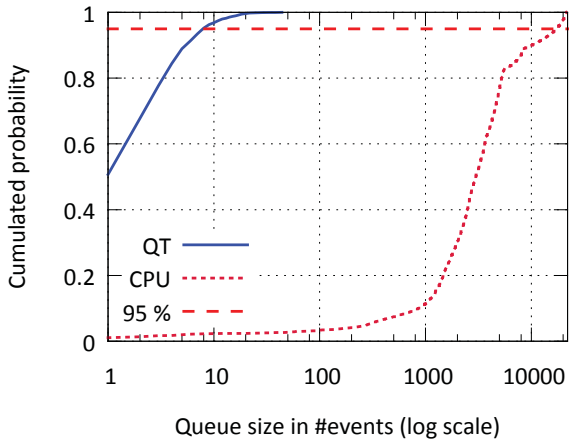
A. System Parameters

All experiments were performed on a computing cluster consisting of 6 physical hosts with 8 cores (Intel(R) Xeon(R) CPU E5620 @ 2.40GHz) and 24 GB memory that are connected by 10-Gigabit-Ethernet connections. Up to 4 operator instances are deployed on one host. We assume that it takes 60 seconds to deploy a new operator instance (TH = 60 seconds), which corresponds to worst case boot times in the Amazon EC2 Cloud[4].

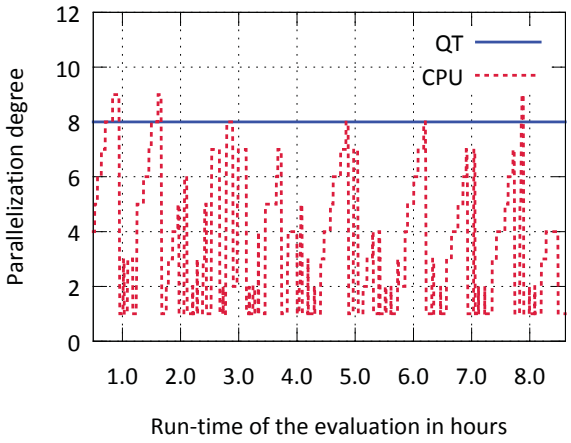
B. Dynamic Degree Adaptation

1) Traffic Monitoring Scenario: Consider a scenario from Traffic Monitoring Systems (TMS) as depicted in Figure 6. On highways, it is often desirable to establish an overtaking ban, especially in danger spots like road construction sites. Given two sensors with synchronized clocks deployed at the beginning and end of the no-passing zone (`pos_1` and `pos_2`), the operator ω detects when a vehicle overtakes another one,

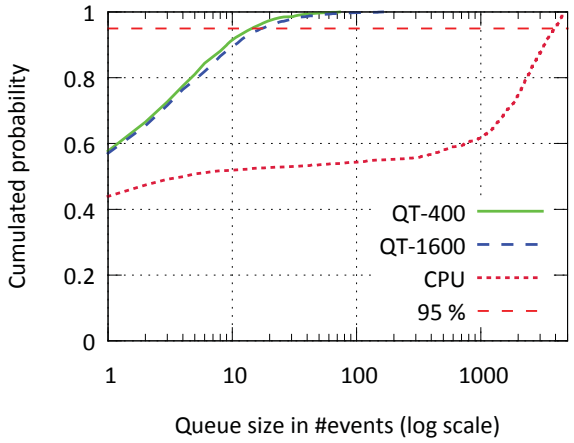
[4]http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ComponentsAMIs.html



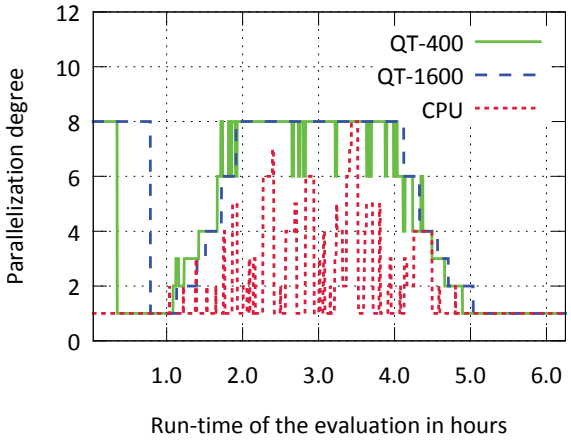
(a) Queue sizes of the QT and CPU approaches under static average inter-arrival time.



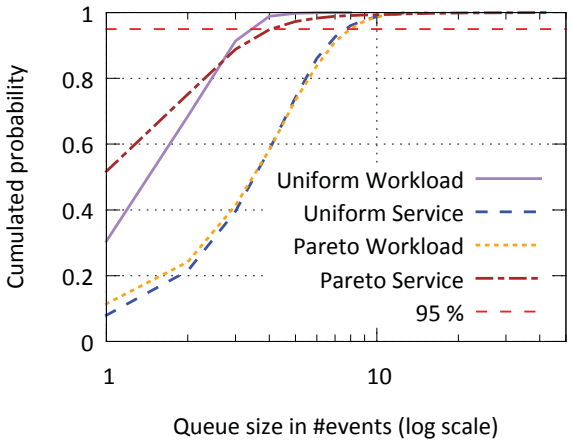
(b) Parallelization degree of the QT and CPU approaches under static average inter-arrival time.



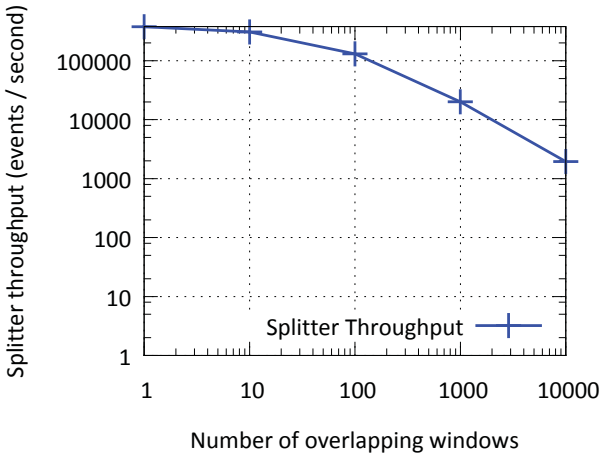
(c) Queue sizes of the QT and CPU approaches under dynamic average inter-arrival time.



(d) Parallelization degree of the QT and CPU approaches under dynamic average inter-arrival time.



(e) Queue sizes at Uniform and Pareto distributions that are approximated.



(f) Splitter throughput in a tuple-based sliding window with different overlap.

Fig. 8. Evaluation results for different scenarios.

requiring timeliness and consistency of event detection. Each time a vehicle passes one of the camera-based sensors, a source event is created. To detect the violations, ω utilizes a selection window: Whenever a vehicle a passes `pos_1`, a window is opened, and when the same vehicle passes `pos_2`, the window is closed. Another vehicle b that appears in the `pos_1` stream within that window has passed `pos_1` *after* a. When b appears again in the same window in the `pos_2` stream, it has passed `pos_2` *before* a. If this is the case, b has overtaken a and thus violated the traffic rules.

Note that the windows span over a dynamic number of events based on the occurrence of start and end events, which is only possible with pattern-sensitive stream partitioning. We assume the following parameters of the setup: Between `pos_1` and `pos_2` (cf. Figure 6), there is a distance of 15 kilometers. Following the speed limit, the vehicles will usually drive 60 kilometers per hour in average; however, there is a ratio of 10 % of faster vehicles on the road which drive between 60 and 72 kilometers per hour (uniformly distributed). It has been shown in a number of measurement studies that the distribution of vehicles on a road is best modeled using a Poisson process [33], thus resulting in exponential distribution of inter-arrival times of vehicles at the checkpoints.

2) Operator Profiles: Fig. 7 shows the measured operator profiles with regard to our infrastructure. We only consider events of type `pos_2`, because events of type `pos_1` are just added to a list which is a negligible operation while events of type `pos_2` are compared to all events of type `pos_1` that have occurred in the selection, leading to noticeable processing times. The processing time grows polynomially with the event rate. For utilizing QT, we approximate the profile with the deterministic values of the 99-percentile and model the queuing system as an M/D/c queue.

Taking into account the operator profiles, the requirements on the queue length for the following evaluations are set to $BL_\omega = 15$ and $P_{required} = 95\%$, yielding buffering delays of less than 20 seconds in 95% of the time even at the highest traffic density, which allows for instance for direct feedback to a driver who violated the traffic rules. In our experiments, we choose $sfreq = slength$.

3) Fixed average inter-arrival time: Figure 8(a) shows the measured queue sizes of our QT-based approach compared to a reactive approach that adapts the parallelization degree depending on the average CPU load of active operator instances: Following Fernandez et al. [20], we add a new instance once the average CPU load over two subsequent time frames of 5 seconds is higher than 70 % and remove one instance when it is less than 50 %. In the QT approach, we used naive forecast [26], using the latest measured workload distribution as the predicted one. In the scenario, the inter-arrival time of cars has an average of 200 ms and the queue sizes are measured in 10 seconds intervals over a run time of more than 8 hours. As can be seen in Figure 8(a), QT reaches a 95-percentile of 8 events, so that in 95 % of the time the queue length is 8 or lower and BL_ω is kept. The CPU based approach fails to keep BL_ω; the 95-percentile is at more than 17,000 events.

Furthermore, we measured the parallelization degree that is generated in the different adaptation approaches (Figure 8(b)). In the QT-based approach, a stable degree of 8 is kept, as the workload classification reliably detects the workload distribution. The CPU-based approach however does not stabilize the degree. Instead, it is gradually increased to a certain level and then suddenly drops, taking another 30 minutes to increase it again. One reason is that the traffic monitoring operator works on large, overlapping partitions. The processing effort per event depends on the number of partitions that comprise the event as well as the number of events that already have been processed within a partition, as a new event from `pos_2` needs to be compared to a growing number of events from `pos_1` in a growing partition. Initially, the CPU based approach assigns many partitions to a small number of instances, because the CPU load in the beginning of processing a partition is still small. When the CPU load grows, it is already too late to add new instances, because the partitions that cause the overload are already assigned to the small number of instances. The algorithm adds more and more instances, and finally overshoots. Then, the same cycle starts again. This is an inherent problem of the CPU based approach in the scenario and similar results are observed for other CPU threshold values.

4) Dynamic average inter-arrival time: In the second scenario, the average inter-arrival time of events grows and shrinks over time to simulate a rush hour. The system is set up with an initial parallelization degree of 8. Within 2 hours, the traffic density grows from 0.5 cars per second to 5 cars per second, stays at that peak level for 2 hours and then gradually decreases over the next 2 hours back to 0.5 cars per second. All other assumptions and requirements are the same as in the previous scenario. When applying QT, we have evaluated different sizes of the time slices containing n measurements (denoted QT-n).

Figure 8(c) shows the cdf of the queue sizes for the QT-400, the QT-1600 and the CPU approach, where the 95-percentile is at 14, 17 and 3,900 respectively. Looking into the logs of the QT-1600 approach, higher queue sizes than 15 mostly happen while scaling up, as it takes some time until all 1600 measurements for a time slice are available. Utilizing a smaller number of measurements diminishes the problem while still giving a good estimation of the workload distribution. In Figure 8(d), the parallelization degree of the QT and CPU approaches is depicted. Similarly to the static scenario, we can observe in the QT approach a stable development of the degree that follows the workload while a heavily fluctuating degree occurs in the CPU approach.

C. Approximating distributions

Recap that according to the considerations in Section V-C, workload and processing time distributions that do not yield a stationary queue length distribution in a queuing system are approximated by exponential or deterministic distributions. To analyze the effects of such approximations, we consider two distributions: The Pareto distribution representing heavy tailed, skewed distributions and the uniform distribution representing short tailed distributions. We run two experiments for each distribution: In the first experiment, the workload follows the approximated distribution while the service time follows an exponential distribution. In the second experiment, the workload follows an exponential distribution while the service time follows the approximated distribution.

1) Uniform distribution: For the uniform distribution, we set the interval to $[100, 200]$. In the first experiment (Uniform

Workload), the given service time is exponentially distributed with an average value of 300 ms. For the inter-arrival time, we approximate the uniform distribution with an exponential distribution such that its cdf is higher than the uniform cdf for the values ≤ 0.99, which is the case at a mean value of 43.21 ms. This results in an optimal parallelization degree of 10. In the second experiment (Uniform Service), the given inter-arrival time is exponentially distributed with a mean value of 40 ms. For the service time, we approximate the uniform distribution with a deterministic distribution of 199 ms. This results in an optimal parallelization degree of 6.

2) Pareto distribution: For the Pareto distribution, we set $x_{min} = 50\ ms$ and $k = 2$. In the first experiment (Pareto Workload), the given service time is exponentially distributed with an average value of 300 ms. For the inter-arrival time, we approximate the Pareto distribution with an exponential distribution such that its cdf is higher than the Pareto cdf for the values ≤ 0.99, which is the case at a mean value of 66.67 ms. This results in an optimal parallelization degree of 6. In the second experiment (Pareto Service), the given inter-arrival time is exponentially distributed with a mean value of 66.67 ms. For the service time, we approximate the Pareto distribution with a deterministic distribution of 500 ms. This results in an optimal parallelization degree of 10.

BL_ω is kept in all experiments (cf. Figure 8(e)), so that the approximations show good results.

D. Splitter Throughput

In the splitter, the predicate logic for typical operators consists of only a fixed number of parameter comparisons that are done for each event in P_s and P_c. What scales up the effort in evaluating the predicates is the number of concurrently open selections, as for each open selection, P_c is evaluated on each event. We analyze this effect by evaluating the sliding tuple-based window of 10,000 events with an increasing number of overlapping windows between 1 and 10,000. Figure 8(f) shows that the Splitter yields high throughput that degrades proportionally to the number of overlapping selections.

VII. Related Work

In the following, we discuss the state of the art in two categories: (i) Operator parallelization approaches and (ii) approaches that aim to adapt operator configurations to keep a buffering limit.

A. Operator parallelization

In *intra-operator parallelization*, internal processing steps that can be run in parallel are identified by deriving operator states and transitions from the query [1], which offers only a limited achievable parallelization degree depending on the number of variables in the query. In the field of *data parallelization* through stream partitioning [18], where incoming event streams are split into independently processable parts that are processed in parallel by replicated instances of the operator, two different partitioning models have been proposed: key-based and batch-based. *Key-based* partitioning, that is partitioning by a key that is encoded in an event, is applied in almost all current data parallelization approaches [10]–[17]. To allow for consistent partitioning, the parallelization degree is limited to the number of different key values, if a common key is available at all, and the membership of a distinct event to a certain pattern must not depend on the occurrence of other events, which limits the expressiveness of the model. *Batch-based* stream partitioning, as proposed in the run-based parallelization approach in [1], proposes to split the streams into batches that are large enough to fit any match to a queried pattern. This can cause communication overhead, as the partitions are always set for the largest possible pattern, and the approach is insufficient to support consistent partitioning for operators that detect patterns of an unknown size, as given for example in aperiodic, periodic and sequence operators [19]. None of those parallelization approaches are suitable for the traffic monitoring scenario in our evaluations.

B. Keeping a buffering limit

The reactive scale-out approach in [20] proposes scaling out based on the current CPU load of operator instances. However, the approach does not offer any guarantees on meeting a specific buffering level, as our evaluation results also have shown. In [21], a more sophisticated parallelization degree adaptation algorithm is proposed that bases on the parameters congestion and throughput. However, it does not guarantee buffer limits on bursty workloads and does not account for the problem of large selections that have a delayed effect on the operator, as is the case in our traffic monitoring operator. Balkesen et al. [34] try to forecast the exact event arrival rate and assume a fixed per-tuple processing cost when determining the optimal parallelization degree, which does not always hold, as we have shown with the operator profiles in our traffic monitoring scenario. Performance modeling approaches, e.g., Queuing Petri Nets [35] or latency estimation metrics like Mace [36], require a deeper knowledge of and interference with the operators and do not provide methods to adapt the parallelization degree in order to yield a limited buffering level at fluctuating workloads. Heinze et al. [37] tackle the problem of latency peaks when moving operator state at scaling out, which does not apply to our approach that splits the streams into independently processable partitions.

VIII. Conclusion

In this paper we have identified two important shortcomings on the way towards timely event detection in CEP systems. First, the state of the art lacks consistent parallelization models for a huge class of operators. Second, state-of-the-art systems are not equipped with a method to adaptively determine the optimal parallelization degree at fluctuating workloads in order to guarantee a buffering limit is met.

To this end, the proposed pattern-sensitive stream partitioning method supports the consistent parallelization of a wide class of important CEP operators. Furthermore, the proposed degree adaptation method is able to meet probabilistic buffering limits at highly fluctuating workloads. Future work will aim at developing a latency model for operators comprising also communication and processing delays and tackling the dependencies between operators in operator graphs.

Acknowledgments

The authors would like to thank the reviewers as well as B. Ottenwälder and S. Bhowmik for their helpful comments.

REFERENCES

[1] C. Balkesen, N. Dindar, M. Wetter, and N. Tatbul, "Rip: run-based intra-query parallelism for scalable complex event processing," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*, ser. DEBS '13, 2013, pp. 3–14.

[2] B. Ottenwälder, B. Koldehofe, K. Rothermel, K. Hong, D. Lillethun, and U. Ramachandran, "Mcep: A mobility-aware complex event processing system," *ACM Trans. Internet Technol.*, vol. 14, no. 1, pp. 6:1–6:24, Aug. 2014.

[3] B. Ottenwälder, B. Koldehofe, K. Rothermel, and U. Ramachandran, "Migcep: Operator migration for mobility driven distributed complex event processing," in *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, ser. DEBS '13, 2013, pp. 183–194.

[4] B. Ottenwälder, B. Koldehofe, K. Rothermel, K. Hong, and U. Ramachandran, "Recep: Selection-based reuse for distributed complex event processing," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14, 2014, pp. 59–70.

[5] B. Koldehofe, B. Ottenwälder, K. Rothermel, and U. Ramachandran, "Moving range queries in distributed complex event processing," in *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '12, 2012, pp. 201–212.

[6] S. Srinivasagopalan, S. Mukhopadhyay, and R. Bharadwaj, "A complex-event-processing framework for smart-grid management," in *2012 IEEE International Multi-Disciplinary Conference on Cognitive Methods in Situation Awareness and Decision Support (CogSIMA)*, March 2012, pp. 272–278.

[7] G. G. Koch, B. Koldehofe, and K. Rothermel, "Cordies: Expressive event correlation in distributed systems," in *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '10, 2010, pp. 26–37.

[8] G. Cugola and A. Margara, "Processing flows of information: From data stream to complex event processing," *ACM Comput. Surv.*, vol. 44, no. 3, pp. 15:1–15:62, Jun. 2012.

[9] C. Clark. (2014) Improving speed and transparency of market data. Http://exchanges.nyx.com/cclark/improving-speed-and-transparency-market-data.

[10] A. Martin, A. Brito, and C. Fetzer, "Scalable and elastic realtime click stream analysis using streammine3g," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14, 2014, pp. 198–205.

[11] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, ser. NSDI'10, 2010, pp. 21–21.

[12] A. Brito, A. Martin, T. Knauth, S. Creutz, D. Becker, S. Weigert, and C. Fetzer, "Scalable and low-latency data processing with stream mapreduce," in *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, ser. CloudCom '11, 2011, pp. 48–58.

[13] D. Battré, S. Ewen, F. Hueske, O. Kao, V. Markl, and D. Warneke, "Nephele/pacts: a programming model and execution framework for web-scale analytical processing," in *Proceedings of the 1st ACM symposium on Cloud computing*, ser. SoCC '10, 2010, pp. 119–130.

[14] (2014, September) Storm. Http://storm-project.net/.

[15] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed stream computing platform," in *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, ser. ICDMW '10, Dec., pp. 170–177.

[16] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, ser. EuroSys '07, 2007, pp. 59–72.

[17] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu, "Auto-parallelizing stateful distributed streaming applications," in *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 53–64.

[18] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A catalog of stream processing optimizations," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 46:1–46:34, Mar. 2014.

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

[20] R. Castro Fernandez, M. Migliavacca, E. Kalyvianaki, and P. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '13, 2013, pp. 725–736.

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

[22] B. Koldehofe, R. Mayer, U. Ramachandran, K. Rothermel, and M. Völz, "Rollback-recovery without checkpoints in distributed event processing systems," in *Proceedings of the 7th ACM international conference on Distributed event-based systems*, ser. DEBS '13, 2013, pp. 27–38.

[23] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: Semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, Jun. 2006.

[24] R. Adaikkalavan and S. Chakravarthy, "Seamless event and data stream processing: Reconciling windows and consumption modes," in *Proceedings of the 16th International Conference on Database Systems for Advanced Applications - Volume Part I*, ser. DASFAA'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 341–356.

[25] D. Gross, J. Shortle, J. Thompson, and C. Harris, *Fundamentals of Queueing Theory*, ser. Wiley Series in Probability and Statistics. Wiley, 2011.

[26] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn, "Self-adaptive workload classification and forecasting for proactive resource provisioning," *Concurrency and Computation: Practice and Experience*, vol. 26, no. 12, pp. 2053–2078, 2014.

[27] D. G. Feitelson, "Workload modeling for computer systems performance evaluation," book draft, to be published by Cambridge University Press in 2015. Version 1.0.1, typeset on April 8, 2014.

[28] M. A. Stephens, "Edf statistics for goodness of fit and some comparisons," *Journal of the American statistical Association*, vol. 69, no. 347, pp. 730–737, 1974.

[29] T. M. Franke, T. Ho, and C. A. Christie, "The chi-square test: Often used and more often misinterpreted," *American Journal of Evaluation*, vol. 33, no. 3, pp. 448–458, 2012.

[30] S. K. Bose. (2014) M/m/m/inf queue. Http://nptel.ac.in/courses/117103017/6.

[31] H. Tijms, "New and old results for the m/d/c queue," {*AEU*} - *International Journal of Electronics and Communications*, vol. 60, no. 2, pp. 125 – 130, 2006.

[32] V. Guerriero, "Power law distribution: Method of multi-scale inferential statistics," *Journal of Modern Mathematics Frontier (JMMF)*, vol. 1, pp. 21–28, 2012.

[33] R. Mao and G. Mao, "Road traffic density estimation in vehicular networks," in *Proceedings of the 2013 IEEE Wireless Communications and Networking Conference*, ser. WCNC '13, April 2013, pp. 4653–4658.

[34] C. Balkesen, N. Tatbul, and M. T. Özsu, "Adaptive input admission and management for parallel stream processing," in *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, ser. DEBS '13, 2013, pp. 15–26.

[35] S. Kounev, "Performance modeling and evaluation of distributed component-based systems using queueing petri nets," *IEEE Transactions on Software Engineering*, vol. 32, no. 7, pp. 486–502, 2006.

[36] B. Chandramouli, J. Goldstein, R. Barga, M. Riedewald, and I. Santos, "Accurate latency estimation in a distributed event processing system," in *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering*, ser. ICDE '11, 2011, pp. 255–266.

[37] T. Heinze, Z. Jerzak, G. Hackenbroich, and C. Fetzer, "Latency-aware elastic scaling for distributed data stream processing systems," in *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, ser. DEBS '14, 2014, pp. 13–22.