Window-Based Data Parallelization in Complex Event Processing

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

vorgelegt von

Ruben Daniel Mayer

aus Heilbronn-Neckargartach

Hauptberichter: Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel

Mitberichter: Prof. Dr. Umakishore Ramachandran

Tag der mündlichen Prüfung: 19.03.2018

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

2018

Acknowledgements

Performing the research that has led to this thesis would not have been possible without the help of many people. First of all, I want to thank my doctoral advisor, Prof. Kurt Rothermel, for his support and guidance throughout the whole time. I also want to thank Prof. Umakishore Ramachandran, who gave me a lot of advice over the years and even came all the way from Atlanta to Stuttgart for my thesis defense.

Special thanks go to Boris Koldehofe, who was my post-doc in the first two years of my PhD. He introduced my into the topic of CEP, and also into the research community in that field. Moreover, I thank Adnan Tariq, my post-doc in the second half of my PhD, for his support and hard work. In the end, the work wouldn't have been possible without such great supervisors. I learned a lot from both of you!

More thanks go to all the colleagues at IPVS who made the work in Stuttgart very pleasant. Namely, I thank Beate Ottenwälder, Christian Mayer, Christoph Dibak, Frank Dürr, and Ahmad Slo, who in one or the other way collaborated with me or gave valuable feedback. Also, I thank Enrique Saurez and Harshit Gupta from EPL at Georgia Tech, who made my stay there a very interesting and enjoyable experience.

Last but not least, I would like to thank the Baden-Württemberg Stiftung and the DFG for their financial support through the projects that enabled the research work done in this thesis.

There is more to a PhD than only good scientific support, a pleasant work environment at university, and the necessary funding. In the course of the PhD, there are ups and downs, and both are better to handle when there is support at home, outside of the office hours. I thank you, An Jing, for your continuous support and encouragement throughout the whole time.

Contents

List of Figures

List of Abbreviations

API	Application Programming Interface
CEP	Complex Event Processing
DAG	Directed Acyclic Graph
FIFO	First In First Out
IaaS	Infrastructure as a Service
iat	Inter-arrival time
IoT	Internet of Things
LAN	Local Area Network
NYSE	New York Stock Exchange
QoS	Quality of Services
QT	Queuing Theory
SPECTRE	SPECulaTive Runtime Environment
TCP	Transmission Control Protocol
VM	Virtual Machine

Abstract

With the proliferation of streaming data from sources such as sensors in the Internet of Things (IoT), situational aware applications become possible. Such applications react to situations in the surrounding world that are signaled by complex event patterns that occur in the sensor streams. In order to detect the patterns that correspond to the situations of interest, Complex Event Processing (CEP) is the paradigm of choice. In CEP, a distributed operator graph is spanned between the event sources and the applications. Each operator step-wise detects event patterns on subsequences, called windows, of its input stream and forwards output events that signal the detection to its successors. To cope with the ever-increasing workload at the operators, operator parallelization becomes necessary. To this end, data parallelization is a powerful paradigm, building on an architecture that consists of a splitter, operator instances and a merger, to scale up and scale out CEP operators. In doing so, the operators need to provide consistent output streams, i.e., not produce false-negatives or false-positives, keep a latency bound on pattern detection, elastically adapt their resource reservations to the workload, and be fault-tolerant against node and network failures. Related work has proposed data parallelization techniques that build on splitting the input event streams of an operator either in a key-based, a batch-based or a pane-based way. These approaches, however, only support a limited range of CEP operators.

The goals of this thesis are (i) to support data parallelization for all window-based CEP operators, (ii) to develop adaptation methods such that CEP operators can keep a user-defined latency bound while minimizing costs for computing and networking resources, and (iii) to develop recovery methods that guarantee fault-tolerance at a low run-time overhead.

To this end, the following contributions are made. First, we propose a window-based data parallelization method that is based on the externalization of the operator's window policy to a data parallelization framework. Second, basing on Queuing Theory, we propose a method to adapt the operator parallelization degree at run-time to the workload such that probabilistic bounds on the event buffering in the operator can be

met. Third, we propose a batch scheduling algorithm that is able to assign subsequent overlapping windows to the same operator instance, so that communication overhead is minimized, while a latency bound in the operator instances is still kept. Forth, we propose a framework for parallel processing of inter-dependent windows that is based on the speculative processing of multiple versions of multiple windows in parallel. Fifth, we propose a lightweight rollback recovery method for CEP operator networks that exploits the externalization of the operator window policy to allow for the recovery of an arbitrary number of operators.

Zusammenfassung

Die zunehmende Verbreitung von Datenströmen aus Quellen wie Sensoren im Internet der Dinge macht situationsbewusste Anwendungen möglich. Solche Anwendungen reagieren auf Situationen in der Umgebung, die durch komplexe Ereignismuster in den Datenströmen signalisiert werden. Um Ereignismuster zu erkennen, die den interessanten Situationen entsprechen, wird heute komplexe Ereignisverarbeitung, oder Complex Event Processing (CEP), intensiv genutzt. In CEP wird ein verteilter Operatorengraph zwischen den Ereignisquellen und den Anwendungen, die CEP nutzen, aufgespannt. Jeder Operator detektiert schrittweise Ereignismuster in Teilsequenzen seiner Eingangsströme, die Fenster genannt werden. Erkannte Ereignismuster werden den Nachfolgern im Operatorengraph durch Ausgangsereignisse angezeigt. Um mit den immer weiter zunehmenden Arbeitslasten der Operatoren umgehen zu können, ist deren Parallelisierung notwendig. Zu diesem Zweck ist die Datenparallelisierung ein mächtiges Werkzeug. Sie basiert darauf, anhand einer dreistufigen Architektur, bestehend aus Splitter, Operatorinstanzen und Merger, die Operatoren zu skalieren. Dabei müssen die Operatoren einen konsistenten Ausgangsstrom erzeugen, der keine falsch-negativen und falsch-positiven Ereignisse enthält, eine Latenzschranke in der Ereigniserkennung einhalten, ihre Ressourcenreservierungen elastisch an die Arbeitslast anpassen, und fehlertolerant gegenüber Knoten- und Netzwerkfehlern sein. Verwandte Arbeiten haben Techniken zur Datenparallelisierung vorgeschlagen, die darauf bauen, eingehende Datenströme im Splitter entweder anhand eines Schlüsselwertes, einer Stapelgröße oder in Scheiben, sogenannten Panes, aufzuspalten. Diese Techniken unterstützen jedoch nur eine begrenzte Auswahl von Operatoren.
Die Ziele dieser Dissertation bestehen darin, (i) Datenparallelisierung für alle fensterbasierten CEP-Operatoren zu ermöglichen, (ii) Adaptionsmechanismen zu entwickeln, sodass Operatoren eine nutzerdefinierte Latenzschranke einhalten können, während die Kosten für Rechen- und Netzwerkressourcen minimiert werden, und (iii) Wiederherstellungsmethoden zu entwickeln, die Fehlertoleranz zu geringen Laufzeitkosten garantieren.

Zu diesem Zweck werden die folgenden Beiträge geleistet. Erstens schlagen wir eine fensterbasierte Datenparallelisierungsmethode vor, die darauf basiert, dass die Fensterbewegungen eines Operators einem Datenparallelisierungsframework gegenüber offengelegt werden. Zweitens schlagen wir eine Methode zur Adaption des Operators vor, die auf Warteschlangentheorie beruht und zur Laufzeit den Parallelisierungsgrad des Operators so an die Arbeitslast anpasst, dass probabilistische Grenzen in Bezug auf die Pufferung von Ereignissen im Operator durchgesetzt werden. Drittens schlagen wir einen Schedulingalgorithmus vor, der aufeinander folgende und überlappende Fenster der gleichen Operatorinstanz zuweist, sodass der Kommunikationsaufwand minimiert wird, während eine Latenzschranke in den Operatorinstanzen eingehalten wird. Viertens schlagen wir ein Framework für die parallele Verarbeitung voneinander abhängiger Fenster vor, das auf der spekulativen, parallelen Verarbeitung mehrerer Versionen mehrerer Fenster basiert. Fünftens schlagen wir eine leichtgewichtige Wiederherstellungsmethode für CEP-Operatorennetzwerke vor, die die Offenlegung der Fensterbewegungen eines Operators ausnutzt, um die Wiederherstellung einer beliebigen Anzahl von Operatoren zu ermöglichen.

1

Introduction

In the recent years, we face an unseen boost of *streaming data* becoming available from sensors, social networks, stock markets, and various other sources. For instance, billions of sensors and *smart objects*, i.e., objects with embedded electronics that enable identification, sensing and actuation capabilities, are deployed throughout the globe, collecting data about the physical world. They grow in numbers [Int14] and have the power to enable new applications in the areas of smart homes, smart cities, environmental monitoring, health-care, smart business and security, paving the way towards the *Internet of Things* (IoT) [AIM10, MSPC12]. This bears a huge economical potential: For instance, in the IoT, revenues of 267 billion USD have been predicted for the year 2020 by Forbes [For17] .

The data streams from sources like sensors contain valuable information about the situation in the surrounding world. For instance, sensor data in a traffic monitoring environment can contain information on the current traffic flow. However, the single data elements—also referred to as *events*—like position updates of cars stemming from GPS sensor readings, contain only low level information. In order to gain higher level insights that are valuable for interested parties—such as other vehicles that adapt their route to a traffic jam or traffic operators that open additional lanes on a high-way—the low level source event streams have to be aggregated such that situations of interest can be detected. To this end, a middleware is needed between the event sources and sinks that enables the *consistent* and *timely* detection of situations by integrating and continuously analyzing the low level source event streams. Consistency in situation detection means that neither false-positives nor false-negatives should occur, i.e., neither should a situation be detected that is not signaled in the source event streams, nor

should a situation detection be missed when the source event streams actually signal it. Inconsistent situation detection can lead to wrong decisions, e.g., wrong routing decisions, that cost money and degrade customer satisfaction, or even can cause serious damage. Timeliness in situation detection refers to the time span between the point in time when the last source event that signals a situation of interest has been emitted and the point in time when the corresponding situation is actually detected and signaled to the interested party. Late detections of a situation are, in the worst case, useless for the end user; the acceptable delay between the occurrence of a situation and its detection is application-dependent and can be specified as a *latency bound* by a domain expert.

Complex Event Processing (CEP) [Luc01, BOO09, CM10, KKR10, CM12c] is a key paradigm that helps in realizing a middleware for situation detection. CEP allows for specifying a network of multiple dependent operators that step-wise transform low-level event streams into complex events that correspond to the situations of interest, e.g., critical power grid situations or traffic jams. Operators detect event patterns on their incoming event streams. Often, the pattern detection is window-based, i.e., patterns are detected on restricted event sequences—denoted as windows—of the incoming event streams. Windows can have arbitrary size and slide—i.e., move—depending on the *window policy* and on the events occurring in the incoming streams. In particular, subsequent windows can overlap, i.e., they have an event sequence in common. CEP systems further provide means to execute the operators in a distributed manner to ensure an efficient utilization of the available resources [PLS$^+$06, RDR10, CGLPN16].

In doing so, applications often encompass a high and fluctuating number of events to be processed. This challenges CEP systems in several ways: First, high rates of incoming event streams make it necessary to run single operators in a parallel fashion, exploiting multi-core architectures (scale-up) as well as computing clusters that encompass multiple machines (scale-out). This challenge is referred to as *operator parallelization*. Second, the fluctuation of data rates makes it necessary to elastically provide the computational resources that are necessary to process the current load, as providing for the worst-case event rates would lead to costly over-provisioning in times of lower pressure. This challenge is referred to as *operator elasticity*. Furthermore, the CEP system needs to be able to cope with node and network failure, referred to as *reliability*.

Generally, *data parallelization* is a promising method to scale up and scale out operators in an elastic manner by employing a split–process–merge architecture. A *splitter* partitions the incoming event streams of an operator into independently processable parts which are processed in parallel by an elastic set of operator copies, denoted as

operator instances. The detected complex events are emitted by the operator instances to a *merger* that sorts them into a deterministic order. In this setting, the crucial question is how to split the event streams such that the partitions can be processed independently by the operator instances in parallel. The existing splitting approaches, key-based [Hir12, FMKP13], batch-based [BDWT13] and pane-based [BT11, KWF+16] splitting, are insufficient in terms of expressiveness, i.e., they do not support window-based CEP operators.

Besides that, window-based operators challenge existing elasticity methods. When windows are large, assigning a window to an operator instances impacts the processing load on that instance for a long time. Moreover, the processing load that a single event in a window puts on the operator instance may depend on the event's position in the window [MTR17]. While processing a window, processing state is gathered in the operator instance. This state may grow over the course of the window, such that later events in the window yield heavier processing load. Elasticity methods that just react on feedback parameters, e.g., CPU utilization in the operator instances [FMKP13], are not able to stabilize the system in such challenging conditions.

Furthermore, the scheduling of windows to operator instances influences the processing load and hence, the latency, induced in the operator instances. Assigning different overlapping windows to different operator instances implies that events from the overlapping sequence of the event stream have to be replicated to both operator instances, leading to higher communication overhead. If the overlapping windows are all assigned to a single operator instance in one batch, events are not replicated and communication overhead is reduced, but the load on that operator instance is higher, so that the latency increases. In the literature, there is a lack of methods to control the trade-off between communication overhead and latency in operator instances.

Lastly, window-based operators lack efficient reliability-preserving mechanisms. Providing active [Sch90, VKR11] or passive [BMST93] standby operators causes a lot of cost; to survive f operator failures, $f+1$ replicas of an operator need to be provisioned. To overcome the large overhead, roll-back recovery has been proposed, where checkpoints of the operator state are periodically taken; in case of a failure, the operator state is recovered from the checkpoint [EAWJ02, SM11]. Still, as window-based operators may gather a large internal state while processing large windows, checkpointing causes a lot of overhead even at failure-free run-time. A more lightweight roll-back recovery method that exploits the window policies in the operator to avoid costly checkpoints of the internal state is still lacking.

The goal of this thesis is to develop concepts and algorithms that support the scalability, elasticity, and reliability of window-based CEP operators. To this end, an expressive stream partitioning method is presented that allows for utilizing data parallelism in all window-based CEP operators. Further, a model-based elasticity method is presented that allows for taking into account workload fluctuations and processing latency fluctuations in operator instances by providing a problem formulation and adaptation algorithm that are based on Queuing Theory. To control the scheduling of windows to operator instances, a batch scheduling controller is proposed that tries to minimize communication overhead while keeping a latency bound in the operator instances. Lastly, a rollback recovery approach is developed that takes into account operator feedback about the window policies in order to avoid heavy-weight checkpointing of operator state, thus reducing run-time overhead.

1.1 Background

This section introduces the background of the thesis work. First, the technological trends that influence the thesis work are discussed in Section 1.1.1. Following that discussion, in Section 1.1.2, the paradigm of CEP is introduced, providing an overview of current CEP systems and discussing requirements posed on the CEP systems in the context of the discussed technological trends and scenarios. High data rates can only be handled by employing parallelization on the CEP system; the current parallelization methods are, hence, discussed in Section 1.1.3 and shortcomings in supporting the requirements on CEP are highlighted. Overcoming those shortcomings for window-based CEP operators is the major goal of this thesis. The thesis has been carried out in the scope of two subsequent projects, funded by the Baden-Württemberg Stiftung gGmbH and the Deutsche Forschungsgesellschaft DFG. Section 1.1.4 provides background information on those research projects.

1.1.1 Technological Trends

Two major technological trends that influence the field of CEP are regarded in this thesis: (1) The surge of *streaming data* being available, and (2) the *cloud computing* paradigm enabling the elastic provisioning of resources. In this section, these trends are discussed with focus on their impact on the design of CEP systems.

Streaming Data

There is an ongoing trend of increasing availability of all kinds of sensor data. In the course of the proliferation of the Internet of Things, "smart" things equipped with sensors are deployed in billions in the surrounding world [Int14], producing a huge amount of streaming data. Moreover, with the surge of social networks such as Facebook, Twitter, etc., the trend of *Social Sensing* is emerging [AA13, MGSR17a], where humans serve as sensor carriers or sensors themselves, emitting messages, posts and tweets. Such data streams contain valuable high-level information that can be exploited by analyzing the low-level data "on the fly". For instance, analyzing sensor streams of cars allows for live vehicle diagnosis [SZG10], analyzing smart meter data allows for load predictions and outlier detections [JZ14], analyzing smart factory data allows for detecting anomalies in the production process [GJK+17], and analyzing Twitter streams allows for performing sentiment analysis [AXV+11] and even predicting crowd behavior [Kal14].

Cloud Computing

Cloud computing [Hay08, BYV+09, AFG+10] is a paradigm that describes the advent of computing and storage as a utility. That means, that users get provisioned with resources *on demand*, and get billed for those resources according to their usage (*pay-as-you-go*). This view of cloud computing is also often referred to as "*Infrastructure as a- Service*" (IaaS). The on-demand resource provisioning in cloud computing enables an *elastic* style of scaling data analytics frameworks according to the fluctuating workload. The incentives of elasticity are clear: Instead of providing (and paying for) resources for the anticipated peak workload, the system rather adapts its resource reservations to the workload in such a way that always enough resources are available to keep the Quality of Services (QoS) goals, such as throughput and latency, but to avoid costly overprovisioning.

Besides the advantages of elastic resource provisioning, cloud computing also offers a high level of abstraction that eases the deployment and management of applications [Hay08]. Instead of administrating and maintaining their own infrastructure, which is cumbersome and costly, users can buy that service as well from the cloud providers. Cloud providers offer a simple resource provisioning abstraction, i.e., predefined "flavors" of virtual machines (VMs). Problems such as load balancing and

placement of the VMs on the infrastructure [RLTB10], migration of VMs [VBVB09], etc., are handled by the cloud provider and are opaque to the user.

1.1.2 Complex Event Processing (CEP)

Complex Event Processing (CEP) [Luc01, BOO09, CM12c] has evolved as the paradigm of choice to detect and integrate events in situation-aware applications. In CEP, domain experts specify a query in a *query language* like Snoop [CM94], Amit [AE04], SASE [WDR06] or TESLA [CM10]. Those languages involve constructs like event sequences, conjunctions, and negations, in order to define the event patterns to be detected. A query corresponds to an event pattern that marks a *situation* of interest occurring in the surrounding world. This overall pattern is broken down into multiple sub-patterns that are detected by a distributed *operator graph* [CBB+03, KMR+13, OMK14, JAA+06, SMMP09]. In the operator graph, each operator detects event patterns on its incoming event streams and produces outgoing events whenever an event pattern was detected. The operators are placed according to a placement strategy [PLS+06, RDR10, CGLPN16] on the available computing nodes between the event sources and sinks.

Examples

In the following, two example scenarios are introduced, where CEP systems are used in order to detect situations of interest from low-level sensor streams: a traffic monitoring system and a "friend finder" application.

(1) Traffic Monitoring. In smart cities, automated traffic monitoring systems are deployed, as depicted in Figure 1.1. On highways, it is often desirable to establish an overtaking ban, especially in danger zones like road construction sites. Given two cameras, Src_1 and Src_2, deployed at two locations at the beginning and end of the no-passing zone (*L1* and *L2*), a CEP system can be used in order to detect when a vehicle overtakes another one. The operator graph consists of 3 operators: Each video stream from the cameras Src_1 and Src_2 is sent to a *number plate detection* operator ω_{plate} that extracts the number plates of distinct cars from the video frames. The detected number plates are streamed to an *overtaking detection* operator $\omega_{overtake}$ that detects when the ordering of number plates in the two incoming streams is different, which indicates overtaking.

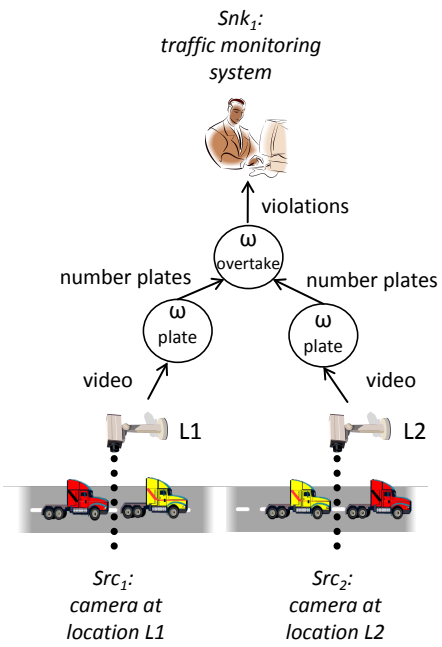


Figure 1.1: Traffic monitoring: Surveillance of a no-passing zone.

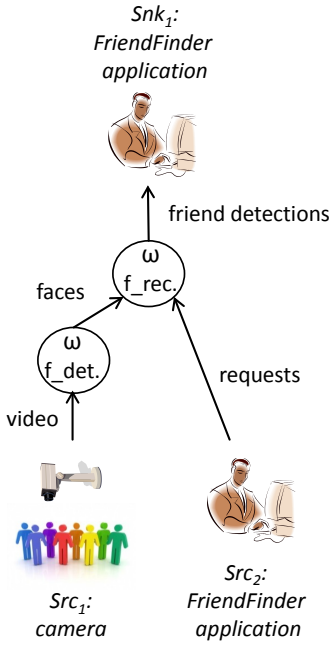


Figure 1.2: Friend finder: Finding a person in a video stream.

(2) Friend Finder. A friend finder application offers the functionality to detect whether a person of interest (e.g., a participant of a marathon) is currently located in a specific area of interest (e.g, the goal of the marathon). To this end, a camera and an operator graph consisting of two operators are employed. The camera Src_1 captures video frames from the area of interest. The video frames are streamed to a *face detection* operator $\omega_{f\_det.}$ which detects all faces of distinct persons in the video stream. On the other hand, the friend finder application is a source Src_2 of request events searching for specific persons of interest. Faces from $\omega_{f\_det.}$ as well as requests from Src_2 are streamed to a *face recognition operator* $\omega_{f\_rec.}$ that detects whether a requested person is currently in the face stream.

Operator Execution

Based on the operator $\omega_{overtake}$ from the traffic monitoring scenario and the operator $\omega_{f\_rec.}$ from the friend finder scenario, some basic concepts of CEP operators are introduced in the following. First, both operators are *window-based*, i.e., they restrict the viable sets of events that can build a queried pattern by employing a sliding window over the incoming event streams. Second, they may further restrict the events that

can build a pattern by employing a *consumption policy*. In order to detail those concepts, $\omega_{overtake}$ and $\omega_{f\_rec.}$ are formalized in the `MATCH-RECOGNIZE` notation [ZWC07], which is briefly explained in the following.

`MATCH-RECOGNIZE` expressions consist of three language clauses. The first language clause, `PATTERN`, describes the search pattern as a sequence of symbols that are further specified in the query's `DEFINE` clause. The `WITHIN ... FROM` clause specifies a window end condition (following `WITHIN`) and window start condition (following `FROM`); both can refer to symbols specified in the `DEFINE` clause. Optionally, a `CONSUME` clause can be used in order to specify consumption policies, as explained later. Note that we have added the `WITHIN ... FROM` clause to the original `MATCH-RECOGNIZE` notation [ZWC07] to increase its expressiveness. It originally stems from the TESLA event specification language [CM10], which is more expressive, but also more verbose than `MATCH-RECOGNIZE`.

The formalization of $\omega_{overtake}$ and $\omega_{f\_rec.}$ is listed in Figure 1.3. To detect violations of the overtaking ban, $\omega_{overtake}$ employs windows: Whenever a vehicle *A* passes *L1* (symbol `A` in the query), a new window *w* is opened, and when the same vehicle passes *L2* (symbol `D` in the query), *w* is closed. This window policy is describe in the pattern's `WITHIN ... FROM` clause. Another vehicle *B* that appears in the *L1* stream within window *w* (symbol `B` in the query) has passed *L1* after *A*. When *B* appears again in window *w* in the *L2* stream (symbol `C` in the query), it has passed *L2* before *A*. If this is the case, *B* has overtaken *A* and thus violated the traffic rules. If *B* appears in the *L2* stream after window *w* has been closed, there is no traffic violation. Hence, the window creates a *context* within which the processing of events is meaningful in detecting the queried pattern. In particular, all cars that pass both *L1* and *L2* within *w* have overtaken vehicle *A* that opened *w*. Similarly, windows are also employed in $\omega_{f\_rec.}$. Each request opens a new window that initiates the context for that request. The window is kept open for a given time span, denoted as the *window scope (ws)*. Within that time span, all face events are compared to the request that opened the window. If a match is found, a "friend detection" event is produced. Please note that multiple windows can overlap. This means that events that are part of multiple different windows are evaluated in the context of each window individually. For instance, a face event that is in the window scope of multiple request event is evaluated for a match to each of those requests.

Overlapping windows can lead to the case that the same event is used in multiple pattern detections. For instance, a face event could be matched to different requests. In some cases, this can lead to ambiguities and is not desired. For instance, in a video

$[\omega_{overtake}]$

PATTERN (A B C)
DEFINE
A **AS** $A.type = L1$
B **AS** $B.type = L1$
C **AS** $C.plate = B.plate$ **and** $C.type = L2$
D **AS** $D.plate = A.plate$ **and** $D.type = L2$
WITHIN *D happens* **FROM** *A*

$[\omega_{f\_rec.}]$

PATTERN (A B)
DEFINE
A **AS** $A.type = request$
B **AS** $B.type = face$ **and** $B.$"`face_match(A)`"
WITHIN *ws* time units **FROM** $A.timestamp$

Figure 1.3: Queries for the example operators $\omega_{overtake}$ from the traffic monitoring scenario and $\omega_{f\_rec.}$ from the friend finder scenario.

surveillance application that detects "tailgating" situations, i.e., when an unauthorized person directly follows an authorized person, there is a one-to-one relation between the unauthorized person's detection and the tailgating situation—the unauthorized person can only directly tailgate one authorized person when it enters the door. After tailgating was detected for an unauthorized person once, no more tailgating events should be produced for that person. To enforce that an event is not used more than once in detecting a pattern, a *consumption policy* can be specified in the query. Consumption policies specify under which conditions an event is *consumed* when being part of a pattern instance, i.e., the event is excluded from further pattern detections. We provide a more detailed discussion of windows and event consumptions in Chapter 2.

Requirements on CEP

Situation-aware applications pose a set of requirements on pattern detection in CEP. The following requirements are discussed in this section: consistency, timeliness, elasticity, and reliability.

Consistency. For CEP systems, it is important that detected events capture the status of the monitored surrounding world in a consistent way, i.e., no events of interest are disregarded (*false-negatives*) as well as no "wrong" events that did not really occur

are delivered to event sinks (*false-positives*). For instance, in the traffic monitoring scenario, all overtaking violations should be detected, as well as no overtaking should be reported that has not occurred. False-negatives and false-positives would undermine the acceptance of an automated traffic control by issuing wrong tickets or leaving transgressors undetected. Similarly, the friend finder application would suffer from false-negatives and false-positives, leading to degraded customer satisfaction.

Timeliness. Besides consistency, timeliness plays an important role in event detection. That means that situations of interest are detected by the CEP system within an acceptable time span from the time of their occurrence until the situation is reported to the event sinks. For instance, in the traffic monitoring application, low latency of overtaking detection allows for direct feedback to drivers who have violated the traffic rules. In the friend finder application, late detection of a person of interest can mean that the relevant person has already left the scene. Depending on the specific needs of the application using the CEP system, latency bounds on situation detection are specified by a domain expert.

Elasticity. Workloads for the operators of a CEP system are often heavily fluctuating as the event rates from the sources change over time. For instance, in the traffic monitoring scenario, the density of cars on the road determines the rate of distinct number plates detected by the ω_{plate} operators, and hence, the incoming event rate of the $\omega_{overtake}$ operator. 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. Similarly, in the friend finder application, over time, a different number of distinct persons are in the detection range of the camera sensor Src_1. This means that the face detection operator $\omega_{f\_det.}$ emits a fluctuating number of face events to the face recognition operator $\omega_{f\_rec.}$, which leads to changing workload. Furthermore, in the course of time, different rates of friend finder requests from Src_2 may be issued, adding to the workload variation of $\omega_{f\_rec.}$. Resource provisioning for peak workloads would block resources even when workloads are low and hence, induce an unnecessarily high cost. With the advent of cloud computing, resources can be elastically provided and billed according to the needs of the user. However, in order to exploit this opportunity to save costs, a CEP system must be able to elastically adapt its resource reservations according to its needs, without compromising consistency or timeliness of event detection.

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

Fault-tolerance. CEP systems should be able to tolerate the failure of computing nodes. That means, that despite of the failure of a number of nodes, the consistency of event detection should not be compromised. At the same time, the recovery mechanism should not induce a lot of overhead at failure-free system run-time.

1.1.3 Operator Parallelization in CEP

High event rates demand that CEP operators are highly scalable. However, sequential operator implementations cannot exploit the power of multi-core architectures and elastic cloud resources. To increase scalability of the CEP system, *operator parallelization* is a necessity, i.e., enabling operators to detect multiple patterns in parallel using multiple cores and even multiple computing nodes. In the following, state-of-the-art operator parallelization methods are analyzed. In doing so, the focus on the analysis is on the scalability, i.e., how much parallelism can be achieved, and the expressiveness, i.e., the range of supported CEP queries, of the parallelization methods. Two main methods are analyzed: Task parallelization and data parallelization.

Task Parallelization

In task parallelization, also known as pipelining or intra-operator parallelization, internal processing steps that can be run in parallel are identified by deriving operator states and state transitions from the query [SGLN+11, BDWT13]. The operator logic is split accordingly, and the identified processing steps are executed in parallel on the incoming event streams. This approach offers only a limited achievable parallelization degree depending on the number of states in the query. For instance, operator $\omega_{overtake}$ from the traffic monitoring scenario in Figure 1.1 only offers four states. Those states are: (1) vehicle *A* detected at *L1*, (2) vehicle *B* detected at *L1*, (3) vehicle *B* detected at *L2*, (4) vehicle *A* detected at *L2*. Hence, using task parallelization, $\omega_{overtake}$ could only utilize four processing units (CPU cores or computing nodes); adding more resources to $\omega_{overtake}$ would not increase the throughput further. This exemplifies that task parallelization only offers limited scalability, which in many cases yields only insufficient operator throughput (cf. [BDWT13]). The same limitations hold true for the face recognition operator $\omega_{f\_rec.}$ from the friend finder scenario. To increase the scalability of operators, a parallelization method that can scale the parallelism *independently* of the query is needed.

A common variant of task parallelization uses *lazy evaluation* techniques on event sequence patterns to increase the operator throughput [CM12b, KSS15]. Those techniques check the event stream for *terminator* events, i.e., the last event of the event sequence in a pattern, and only evaluate preceding events when such a terminator event is found. The underlying assumption is that a terminator event can be determined independently of other events, e.g., solely based on its event type. However, often, sequence patterns depend on the comparison of the events' payload, e.g., a stock quote increasing 3 times in a row; whether a quote is the third in a row that is increasing can only be determined when the two preceding quotes are analyzed. Hence, such techniques are only addressing a subset of possible event patterns.

Data Parallelization

In *data parallelization* [MBF14, HSS+14, CCA+10, BMK+11, BEH+10, SHGW12, BAJR14, BDWT13, MKR15, MMTR16, MMA17, MTR17, DMM17b], instead of splitting the operator logic into different states, the incoming *event streams* are split into partitions that can be processed by a number of identical instances of the operator. Figure 1.4 depicts the architecture of a data parallelization framework, consisting of a splitter, a number of operator instances and a merger. The splitter assigns events from the incoming event streams according to a *partitioning model* to different operator instances. The execution of an operator instance is controlled by a runtime environment (RE). This comprises the management of partitions and communication with the splitter. The RE receives information about the assigned partitions and the corresponding events, and manages the operator execution so that the assigned partitions are processed. The merger ensures that an ordering between all produced events is established if such an ordering is required at subsequent operators or event sinks. In data parallelization, the crucial question is how to split the incoming event streams such that the operator instances consistently detect the event patterns according to the query. To this end, three different partitioning models have been proposed in the literature: key-based, batch-based and pane-based.

Key-based partitioning proposes to split the event stream by a key that is encoded in the events [ZR11, Hir12, FMKP13, Ged14, MMTR16, MMA17], e.g., a stock symbol in algorithmic trading [Hir12] or a post ID in social network analysis [MMTR16]. Different key value ranges are assigned to different operator instances. However, the parallelism is restricted to the number of different key values; moreover, not all pattern definitions in CEP exhibit key-based data parallelism. In many cases, there is no

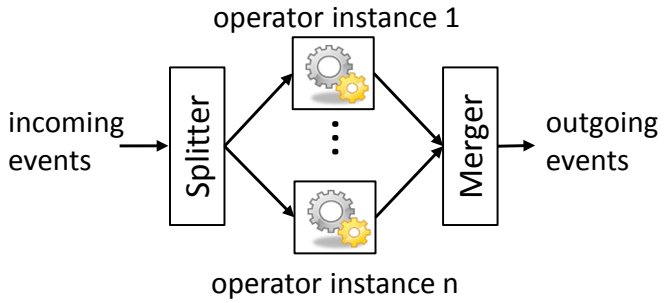


Figure 1.4: Conceptual architecture of a data parallelization framework.

common key available that would allow to group events to different operator instances. Instead, the membership of a distinct event to a certain pattern may depend on the appearance of other events. This is, for instance, the case in the $\omega_{overtake}$ operator from the traffic monitoring scenario. The overtaking situation is detected just when vehicle B is detected a second time before vehicle A. That means, that the appearance of both vehicles A and B is put into a temporal relation. Sending all events of vehicle A to an operator instance $\omega^{A}_{overtake}$ and all events of vehicle B to a different operator instance $\omega^{B}_{overtake}$ would prohibit to detect the overtaking situation.

Batch-based partitioning, as proposed in the run-based parallelization approach in [BDWT13], splits the incoming event streams into batches that are large enough to fit any match to an instance of the queried pattern. Besides the problem that this can cause communication overhead when patterns fluctuate in their size, the approach is insufficient to support partitioning for operators that detect patterns of an unknown size. This is the case in the $\omega_{overtake}$ operator from the traffic monitoring scenario, where a window closes only when the same vehicle that opened the window appears at the second sensor Src_2: How long a window is opened depends on the speed of the vehicle, which might heavily fluctuate over different windows. Also in many other CEP operators, the window size is unknown a-priori, e.g., in aperiodic, periodic and sequence operators as defined in the Snoop pattern definition language [CM94].

Pane-based partitioning has been proposed in stream processing systems [BT11, KWF+16]. For instance, when the maximum or median value of a window of 1 minute shall be computed, that window is split into 6 fragments of 10 seconds, the fragments' maximum or median values are computed in parallel, and the global window's value is aggregated from the fragments' results. This parallel aggregation procedure is based on the idea of pane-based aggregations [LMT+05]. However, CEP patterns often impose a temporal dependency between the events of a window that prevents the vertical

splitting, e.g., when a sequence of events A and B is searched. Furthermore, additional constraints on the events can be formulated, e.g., A and B have a parameter x, such that $A : x > B : x$ (e.g., to detect chart patterns in stock markets [Hir12]). If the events are scattered among different panes, such dependencies and constraints cannot be analyzed.

Concluding from the observations we have made so far, there is currently no consistent operator parallelization method that allows for a high degree of parallelism for window-based CEP operators.

1.1.4 Project Background

This thesis has been carried out in the course of two subsequent projects: (1) "Complex Event Processing in the Large" (CEPiL), a research project funded by the research program "Internationale Spitzenforschung II" of the *Baden-Württemberg Stiftung gGmbH*, and (2) "Parallel Complex Event Processing to Meet Probabilistic Latency Bounds" (PRECEPT), a research project funded by the Deutsche Forschungsgemeinschaft DFG, research grant RO 1086/19-1. It is worth to mention that in both projects, a strong collaboration of the IPVS with the research group of Prof. Umakishore Ramachandran from Georgia Institute of Technology, USA, was established. In this section, the background of both projects is described briefly.

CEPiL

The goal of the CEPiL project, running from 2010 to 2013, was to develop concepts for highly scalable, reliable and secure CEP systems. The outcomes of the project were manifold. Among others, several research papers and 2 doctoral theses [Sch15, Ott16] were published. In particular, the author of this thesis developed a concept for rollback recovery of CEP operators, which is part of this thesis and described in Chapter 6. Here, a short overview of the other results of the CEPiL project is provided.

To achieve high scalability, two research challenges have been addressed in the project. On the one hand, Schilling et al. [SKR11, Sch15] investigated operator placement strategies in heterogeneous and heavily constrained environments. They developed a decentralized placement algorithm that is able to minimize network usage while being adaptive to dynamic changes of processing nodes, rules, and load characteristics. On the other hand, Ottenwälder et al. [KORR12, OKRR13, OKR+14a, OKR+14b, Ott16]

investigated mobility-aware CEP systems. They developed methods to automatically adapt the processing of data streams to the changing location and range-of-interest of the user. In particular, the migration of operators and the sharing of similar query results among multiple users has been investigated.

For increasing the reliability of CEP systems, besides the roll-back recovery approach described in Chapter 6 of this thesis, an active-standby approach has been developed by Völz et al. [VKR11]. In particular, they proposed an algorithm for coordinating the operator replicas such that no false-positives, duplicates and false-negatives occur while the overhead of coordination is minimized.

In terms of security in CEP, Schilling et al. [SKRR13, Sch15] proposed an approach to prevent the inference of confidential input streams from legally received output streams of the CEP system. In their approach, access policies can be specified for each input stream based on a measure of obfuscation; those policies are then enforced by algorithms for access consolidation.

While working on the CEPiL project, it was found that despite of the improvements achieved by operator placement and support for user mobility, the parallelization of operators is a major challenge toward achieving high scalability. As a result of this, the PRECEPT project was devised.

PRECEPT

The goal of the PRECEPT project, started in 2015, is to develop methods for the configuration and dynamic adaptation of operators and operator networks, such that the overall CEP system can probabilistically meet a latency bound under varying and dynamic workloads. To this end, three sub-goals are addressed in PRECEPT. First, expressive data parallelization methods are developed that support window-based CEP operators. Second, methods for the configuration and dynamic adaptation of single CEP operators are developed, such that a single parallel operator can probabilistically meet a latency bound under dynamic workloads. Third, methods for the cost-minimal configurations of a network of operators are developed so that the entire network of operators can meet probabilistic end-to-end latency bounds. The first two goals, expressive data parallelization and dynamic operator configuration, are addressed in this thesis. The third goal, optimization throughout the entire operator graph, is subject to current research in the PRECEPT project that is outside of the scope of this thesis.

1.2 Research Scope and Goals

The overall research goal of this thesis is to support window-based CEP operators with methods for data parallelization, elasticity, and reliability, such that user-defined latency bounds can be met while the overall cost of computation and communication is minimized. This goal is broken down into several sub-goals, which are introduced and detailed in the following.

Expressive data parallelization. Previous work on parallel CEP has dealt with task parallelization or data parallelization that employs key-based, batch-based or pane-based stream partitioning. Those approaches do not support many of the window-based CEP operators that can be specified in expressive query languages such as Snoop [CM94] and Tesla [CM10]. Hence, the first goal of this thesis is to develop a suitable data parallelization method for *all* window-based CEP operators. This requires concepts for consistent event stream splitting and the parallel execution of operator instances.

Adapting the parallelization degree. The rate of events streamed to a CEP operator can change tremendously over time. The second goal of this thesis is to develop methods to adapt the number of operator instances at system run-time. The main challenge for window-based operators is that adaptations have to be planned ahead of time: When windows are large, the assignment of a window to an operator instance influences the processing load imposed on that instance for a long time.

Minimizing communication cost. When splitting the event streams into overlapping windows, the scheduling of those windows to the available operator instances poses a trade-off between load balancing and communication overhead. When overlapping windows are assigned to different operator instances, the events from the overlap are replicated. This means that those events can be processed in parallel, which reduces the processing latency and balances the load between the operator instances. However, replicating the events poses a higher communication overhead between the splitter and the operator instances. The third major goal of this thesis is to control the trade-off, such that a latency bound is kept in the operator instances while the communication overhead between splitter and operator instances is minimized.

Supporting event consumptions. Event consumptions prohibit an event to be part of multiple pattern instances. This poses challenges on data-parallel processing of overlapping windows, as event consumptions can induce inter-window dependencies. The forth goal, hence, is to resolve those dependencies such that inter-dependent windows can be processed in parallel.

Efficient operator recovery. When computing nodes fail, the processing state of operators is lost. The challenge is to allow for efficient state recovery without inducing too much run-time overhead, e.g., for taking checkpoints. The fifth goal in this thesis is to support efficient recovery schemes that work for non-parallelized as well as data-parallel CEP operators.

1.3 Contributions

In this thesis, work presented in [KMR+13], [MKR14], [MKR15], [MTR17] and [MST+17], is combined and extended toward a reliable parallel CEP system. In the course of the research projects CEPiL and PRECEPT, the work has been carried out jointly with colleagues from the IPVS, University of Stuttgart, Germany, and the Georgia Institute of Technology, USA. Hence, the technical contributions of the author of this thesis are highlighted in the following description.

The contributions of this thesis are:

1. An expressive stream partitioning model, referred to as *pattern sensitive stream partitioning*, together with an implementation in a data parallelization framework. The partitioning model allows to consistently parallelize a wide class of CEP operators and ensures a high degree of parallelism. In particular, it comprises an interface to externalize the operator's window policy. This work has been published in [MKR14] and [MKR15]. The main contributions of the author of this thesis are the window-based stream partitioning model, the expressive interface definition to the operator's window policy and the implementation and evaluation of the overall data parallelization approach.

2. A model-based proactive controller, basing on Queuing Theory (QT), to adapt the operator parallelization degree to the workload fluctuations, such that a bound on the buffering induced in an operator can be enforced. This work has been

published in [MKR14] and [MKR15]. The main contributions of the author of this thesis are the development, implementation and evaluation of the QT-based parallelization degree adaptation model.

3. A scheduling algorithm and controller to minimize communication cost when assigning overlapping windows to operator instances. The controller is based on a model that predicts latency peaks in operator instances, such that the optimal amount of overlapping windows can be scheduled to each operator instance dynamically at run-time. This work has been published in [MTR17]. The main contributions of the author of this thesis are the model-based batch scheduling controller, the predictive latency model and the implementation and extensive evaluation of the controller and of the latency model.

4. A framework, named SPECTRE, for parallel processing in the face of event consumptions. To overcome inter-dependencies between overlapping windows, the SPECTRE framework employs a speculative processing method that allows the execution of multiple versions of multiple windows using different event sets in parallel. This work has been published in [MST+17]. The main contributions of the author of this thesis are a probabilistic model that predicts the survival probability of window versions based on feedback about partial pattern matches from the operator instances and a scheduling algorithm that ensures that always those window versions are processed that have the highest survival probability.

5. A rollback recovery method for CEP operator networks, denoted as *savepoint recovery*. The proposed method avoids heavy-weight checkpointing of operator state. The basic idea is that in between the processing of two different windows, the processing state of the operator is empty, so that the overall operator state only depends on the positions of the next window in the incoming event streams. Savepoint recovery relies on the externalization of the window scopes from the operator to the recovery system, and the coordination of multiple such scopes on multiple adjacent operators in order to capture a consistent state of the whole operator graph. This work has been published in [KMR+13]. The main contributions of the author of this thesis are the savepoint updating and coordination algorithm in the operators, the failure recovery algorithm that restores operator states, and the implementation and evaluation of the approach in a simulation framework.

1.4 Structure

The structure of the thesis is organized as follows. Chapter 2 introduces the fundamental assumptions in this thesis, describing an event processing and a system model as well as the basic architecture of a data parallelization framework for CEP operators. In Chapter 3, the data parallelization framework is detailed by describing its pattern sensitive stream partitioning model and the QT-based parallelization degree adaptation method. The batch scheduling controller and algorithm are introduced in Chapter 4, describing how the maximal amount of overlapping windows can be scheduled to a single operator instance while a latency bound in that instance can still be kept. The problem of inter-window dependencies that are caused by event consumptions is tackled in Chapter 5, describing the SPECTRE system that resolves dependencies by means of speculative execution. Finally, the topic of fault tolerance is handled in Chapter 6, introducing the light-weight savepoint recovery algorithm that allows for the recovery of multiple failed operators. Finally, Chapter 7 closes this thesis by providing a summary and an outlook to new trends and open research questions in the field.

2

Fundamentals

This chapter introduces a general system and CEP model. Further, it introduces a generic data parallelization architecture for CEP operators. Parts of the system model and architecture description have already been published in [KMR+13], [MKR14], [MKR15], [MTR17] and [MST+17]. The notation and system model of those publications is unified and refined here.

2.1 CEP Model

This section introduces the general models and notations of events, queries, and CEP operators used throughout this thesis.

Operator Graph. The operation of a CEP system is modeled by a directed, acyclic graph (DAG)—the *operator graph* $G(Src \cup \Omega \cup Snk, L)$—that interconnects sources in *Src*, operators in Ω, and sinks in *Snk* in form of event streams in $L \subset (Src \cup \Omega) \times$

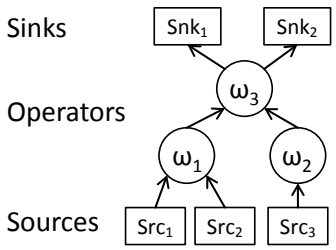


Figure 2.1: Schematic of an exemplary operator graph.

$(\Omega \cup Snk)$. In this model, the sources act as producers of basic events (e.g., sensor streams), operators perform correlations on their incoming event streams to produce new outgoing events, and sinks consume events emitted by the CEP system. The functionality of each operator is defined by a distinct *query*. The query contains a set of patterns to be searched in the incoming event streams of an operator. There are many different CEP query languages that differ in their expressiveness, e.g, Snoop [CM94], Amit [AE04], SASE [WDR06] and TESLA [CM10]. Furthermore, user-defined functions can be embedded in a query, e.g., computer vision algorithms embedded in a face recognition operator.

Figure 2.1 depicts an examplary operator graph, comprising three sources, three operators and two sinks, all represented by vertices, and event streams represented by directed edges. In the following, models of the components of the operator graph, i.e., sources, operators, sinks and event streams, are described in more detail.

Event. An *event* is a data element that represents a change or a condition in the physical world that potentially is of interest to the applications using the CEP system, i.e., to the sinks in the operator graph. Events can be basic events stemming from event sources such as sensors that represent low-level information such as a position update of a car, or more complex events stemming from CEP operators that represent high-level information such as the detection of a traffic jam. Each event consists of *content* and *meta data*. There are no restrictions on the content of events; for instance, an event content can comprise numerical data from sensors, text from a posting in a social network, a video frame captured by a camera, or a notification about the occurrence of a situation of interest emitted by a CEP operator. The meta data of an event consists of the following fields: (1) the event *type*, and (2) a *timestamp*. The event type abstracts common properties of a similar set of events; as such, it is a basic building block of all CEP languages and systems [CM94, AE04, WDR06, CM10]. The timestamp reflects the physical time of occurrence of the situation that caused the event to happen. For instance, if the event reflects a detection of a person in a scene by a face recognition operator, the timestamp of the event signaling the detection may represent the time when the person was captured by the camera in the video frame that eventually led to the detection. An important assumption is that timestamps of events that are produced by the CEP system are computed solely based on the timestamps of the incoming events, but not based on the wall clock time of event production inside of the CEP system. On the other hand, events produced by event sources, such as sensors, may contain timestamps that reflect the wall clock time of the underlying observation (e.g., time of a temperature measurement).

Event Stream. An *event stream* $(p,d) \in L$ is directed from a producer p to a destination d and ensures that events are delivered in the order they are produced. We call p the predecessor of d and d the successor of p. Accordingly, (p,d) is called an *outgoing stream* of p and an *incoming stream* of d. Events from different incoming streams have a well-defined, global ordering that is independent of the physical time of their arrival at the operator.

Event Source. Event sources emit events to the CEP operator graph via event streams that connect the sources to the ingress operators of the graph. Often, event sources are sensors; however, social networks, stock exchanges, and other arbitrary applications can be event sources as well.

Event Sink. Event sinks receive events that are produced by the operator graph via event streams that connect the egress operators to the sinks. Events emitted to sinks signal that a situation of interest has occurred. As such, requirements on the consistency and timeliness of the delivered events are imposed by the event sinks.

Operator. An operator $\omega \in \Omega$ performs processing of its incoming event streams $(in,\omega) \in L$, denoted by I_ω. During its execution, ω conceptually performs a sequence of *correlation steps* on I_ω. In each correlation step, the operator determines a *window* w which is a finite subset of events in each stream of I_ω. A correlation function $f_\omega : \mathsf{w} \rightarrow (e_1,\ldots,e_m)$ specifies a mapping from a window to a finite, possibly empty set of events produced by the operator. The produced events are written in order of occurrence to its outgoing event streams. For each outgoing stream a different set of events may be written.

To express the set of relevant events in a correlation step, the query of an operator imposes a *window* of valid events on its incoming event streams. Such a window can depend on time or the number of events [WDR06, CM10, CM12a, KMR+13, MKR15, MTR17, DMM17b], but also on more complex predicates, e.g., on the content of events [GMM+16] or (combinations of) specific event occurrences that mark the beginning and end of a window [MKR15]. With the arrival of events in the stream, the window can capture new events, while old events fall out of the window scope. This concept is known as a *sliding* window [ABW06]. In this paper, we denote the valid window at a specific point in time as w_i. When the window slides, the subsequent valid windows are denoted as w_{i+1}, w_{i+2}, etc.

How a window moves over event streams is defined in the *window policy*. If multiple events within a window match a queried pattern, this ambiguity is resolved by a *selection policy*. Furthermore, the ability to re-use events from a pattern detected in one

window in another pattern in a different window can be restricted by a *consumption policy*. In the following, those policies are defined in more detail.

Window Policy. According to the pattern definition, windows can have different sizes and a different number of events can occur between two start events of subsequent windows. We denote the period of time that a window spans, i.e., the time between the first event and the last event of a window, as the *window scope*, *ws*. Further, we denote the period of time between two start events of subsequent windows as the *window shift*, Δ. Both window scope and window shift can be fixed or flexible. Depending on the window scope and shift, different subsequent windows can *overlap*, i.e., events are part of multiple different windows.

Selection Policy. In detecting a pattern within the scope of a window, there can be ambiguities. This is the case when multiple events within the same window match a searched pattern. To resolve this issue, a *selection policy* specifies which of the candidate events to select for building the corresponding complex event. Common selection policies select the earliest or latest candidate events, but also more sophisticated policies are possible.

Consumption Policy. A further ambiguity in pattern detection is whether an event is allowed to be used in multiple pattern instances or not. In some cases, multiple correlations of the same event are problematic. If there is a many-to-one relation between incoming events and detected situations, i.e., many events build a pattern instance but a single event can only be part of one pattern instance, contradicting complex events are produced when events are re-used.

Many-to-one or one-to-one relations are a common case in situation detections, e.g., in a video surveillance application that detects tailgating situations, i.e., when an unauthorized person directly follows an authorized person. This is a one-to-one relation between the unauthorized person's detection and the tailgating situation—the unauthorized person can only directly tailgate one authorized person when she enters the door. After tailgating was detected for an unauthorized person once, no more tailgating events should be produced for that person. Another example is a query from a camera surveillance application that contains the pattern "Person *A* is *directly* followed by Person *B* within 10 seconds". A single Person *A* can only be directly followed by one other Person *B*. After the "directly-followed-by" relation is detected for the first time and a complex event is emitted, the corresponding Person *A* event shall not be part of further "directly-followed-by" relations.

To resolve such issues, event specification languages allow for the specification of a *consumption policy* [CM94, ZU99, AE04, CM10]. The consumption policy defines which selected events are consumed after they have participated in a complex event detection: It might be *none*, *all* or *some* of them—e.g., depending on the event type or other parameters.

Example An operator ω receives two event streams which contain events of type A and B. The query implemented in ω detects when an event of type B follows an event of type A within 1 minute. This query can be formalized in the extended `MATCH-RECOGNIZE` notation [ZWC07] introduced in Section 1.1.2 as follows:

$[Q_E]$

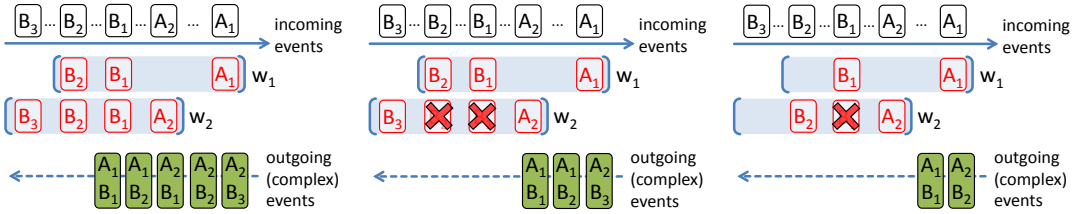
PATTERN (`A B`)
DEFINE
`A` **AS** $A.type = A$
`B` **AS** $B.type = B$
WITHIN 1 minute **FROM** $A.timestamp$

This pattern can be detected by opening a window with a scope of 1 minute (window scope) whenever an A event occurs (window shift); when a B event is detected in a window opened by an A event, a complex event can be created.

Suppose the events A_1, A_2, B_1, B_2 and B_3 occur in the event stream in that order, i.e., A_i denotes the i-th occurrence of an event of type A in the stream (cf. Figure 2.2). Now, different combinations of selection and consumption policies results in different complex events being emitted. In the following, three examples are provided.

Let us assume that the first A in a window is correlated with every B in the same window—this can be defined in the selection policy as "Earliest A, each B". As shown in Figure 2.2a, 5 complex events are detected:[1] $\frac{A_1}{B_1}$, $\frac{A_1}{B_2}$, $\frac{A_2}{B_1}$, $\frac{A_2}{B_2}$, and $\frac{A_2}{B_3}$. Notice, that the corresponding incoming events are correlated multiple times, i.e., they are not consumed after building a complex event. In the example in Figure 2.2b, selected events of type B are consumed when a complex event is detected, referred to as consumption policy "selected B". Now, only 3 complex events are produced: $\frac{A_1}{B_1}$, $\frac{A_1}{B_2}$, and $\frac{A_2}{B_3}$. In that case, B_1 and B_2 are not re-used after being correlated with A_1 in the first window w_1. In a third example in Figure 2.2c, the selection policy is set to "Earliest A, earliest B"

[1] $\frac{X}{Y}$ denotes a complex event created from incoming events X and Y.



(a) SP: Earliest A, each B. CP: None.

(b) SP: Earliest A, each B. CP: Selected B.

(c) SP: Earliest A, earliest B. CP: Selected B.

Figure 2.2: Query Q_E with different selection policies (SP) and consumption policies (CP).

and the consumption policy is set to "selected B". Now, the produced complex events are $\frac{A_1}{B_1}$ and $\frac{A_2}{B_2}$.

As shown with the examples above, the set of produced complex events depends on the interplay between selection and consumption policy.

2.2 System Model

Computing Infrastructure. The operators of a given operator graph G are hosted in a distributed computing infrastructure that consists of a set of n computing nodes connected by a communication network. A node may comprise multiple processing entities (e.g., a processor or processor core), possibly sharing physical memory, and may host multiple operators. Different nodes can communicate via communication channels that are established for each event stream in G and guarantee eventual in-order delivery of streamed events. In Chapters 3 to 5 of this thesis, nodes and communication channels are assumed to be failure-free. In Chapter 6, it is investigated how to detect node and network failures and how to recover the operator graph from such failures.

Administrative Domains. Computing nodes are grouped into disjoint administrative domains. For example, a domain might be a public data center of a cloud provider or a private data center of a company. In this thesis, the following assumptions are made about domains: (1) Nodes belonging to the same domain typically share the same LAN. (2) All resources of a domain belong to the same administrative unit, i.e., all of a domain's resources are controlled and managed by a single provider. In the general case, the execution of an operator graph can be distributed over multiple domains (cf. Figure 2.3). The mapping of operators to domains typically depends on the location of sources and sinks. For example, an operator that significantly reduces the event rate

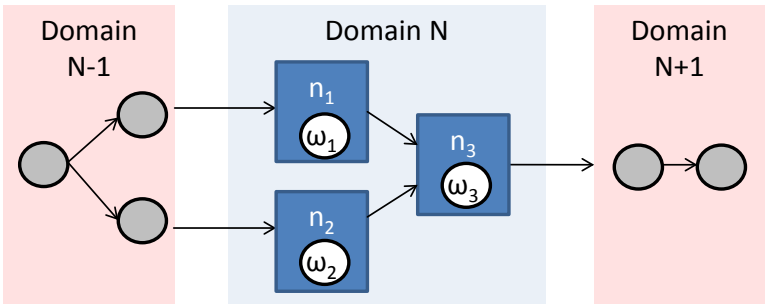


Figure 2.3: Administrative domains. The coarse-grained placement of operator subgraphs on the different domains is given.

by aggregating the output of a source should be located close to that source in order to save network bandwidth. This "coarse-grained" placement problem [CGLPN16] has been subject to extensive research resulting in a wide range of algorithms that allow for minimizing bandwidth usage [PLS+06, RDR10] or dealing with resource constraints [SKR11].

In this thesis, we assume that such a coarse-grained placement of the overall operator graph on the different domains has already been determined. Hence, operator (sub-)graphs that are deployed within a single domain can be considered in isolation. This has the following consequence on the infrastructure model. As the nodes are close to each other and the resources are under control of a single provider, it is assumed that probabilistic bounds on the worst-case communication latency between nodes of the same domain are known. For instance, such a probabilistic bound may be: "the communication latency between node n_1 and n_2 is less than 5 ms in 99% of the time". A practical example of such a latency bound can be found in the PingMesh system [GYX+15] that measured in a Microsoft data center a 99th percentile of inter-node latency of 1.34 ms.

2.3 Data Parallelization Architecture

2.3.1 Overview

To execute an operator ω in a parallel fashion, it is embedded into a *data parallelization framework*. The basic model of such a framework consists of a split–process–merge architecture, as depicted in Figure 2.4. The single components, i.e., splitter, opera-

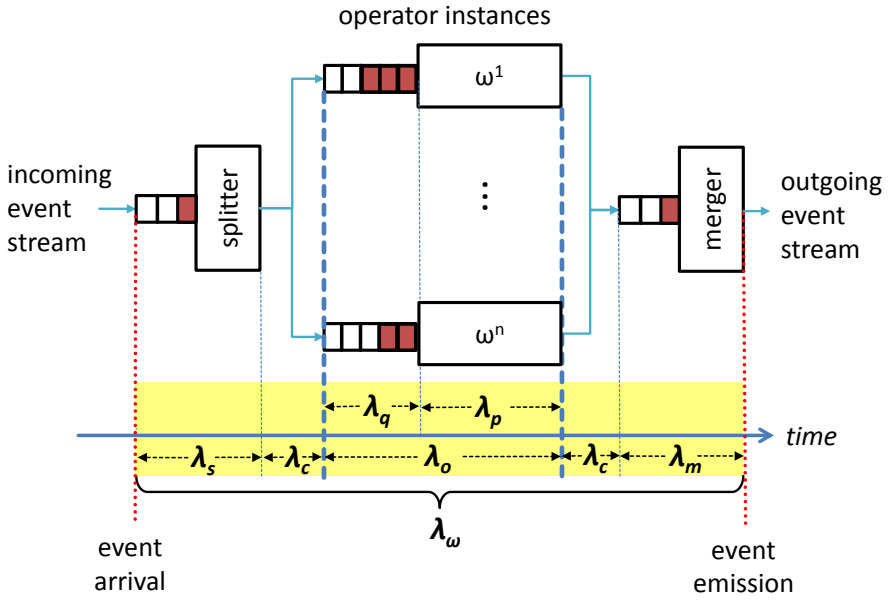


Figure 2.4: Parallel operator model and composition of operator latency between event arrival and (complex) event emission.

tor instances, and merger, are distributed over the available computing nodes of ω's domain. They communicate via event streams as defined in Section 2.1. To support asynchronous operation, the components are each equipped with an incoming event queue. Events arriving from a predecessor component are placed in the queue, and removed after being processed by the respective component. In the following, the tasks of the different components are detailed.

Splitter. The splitter is responsible for partitioning the incoming event streams into windows, according to the operator's window policy (cf. Section 2.1). When the start of a new window is detected, the window is *scheduled* to one of the operator instances. Events that are part of scheduled windows are forwarded to the operator instances accordingly.

Operator Instance. An operator instance processes events from its incoming event stream according to the assigned windows. When a pattern is detected, outgoing events are emitted to the merger. The number of operator instances is elastic, i.e., new instances can be added or existing instances can be removed at system run-time.

Merger. The merger receives all produced events from the operator instances corresponding to the detected patterns. It sorts the events from the different event streams into a well-defined ordering given by their time stamps.

2.3.2 Operator Latency

From the point of view of an event sink, the *situation detection latency* is the period of time from the occurrence of a source event that signals a situation of interest until the situation is actually detected and signaled to the interested sink by a corresponding complex event. As the delayed detection of a situation degrades the benefits for the sink application, it imposes a *situation detection latency bound* on the CEP system. The situation detection latency bound is sub-divided into individual *operator latency bounds* for each single operator. In this thesis, we assume that this division is given, so that each operator has its individual operator latency bound assigned. The goal is to configure the data parallelization framework in such a way that a given operator ω keeps its given operator latency bound LB_ω. An important assumption in this thesis is that all latency bounds are probabilistic. That means that in P_{LB} percent of cases, the latency bound of LB_ω time units must be kept. This assumption is practical in real-world scenarios, as reaching a 100 percent latency guarantee may be very expensive or even impossible due to unpredictable workload fluctuations.

Latency occurring within a parallelized operator is composed of queuing latency, processing latency and communication latency in or between the different operator components (cf. Figure 2.4). Recall that, following the assumption of placing all components of an operator in a single domain (cf. Section 2.2), the communication latency between the components is bounded and known. Then, the overall operator latency λ_ω is calculated as:

$$\lambda_\omega = \lambda_s + \lambda_c + \lambda_o + \lambda_c + \lambda_m$$

with λ_s being the latency of the splitter, λ_o the latency of the operator instances, λ_m the latency of the merger, and λ_c the communication latency between two different components.

It is assumed that bounds on λ_s and λ_m are known based on profiling the splitter and merger component. This is a reasonable assumption, as splitting and merging are light-weight operations that should not become the system bottleneck. Hence, λ_o is considered as the share of λ_ω that is in control of the configuration of the parallel operator—a detailed definition of *operator configuration* will be provided in Section 2.3.3. In an operator instance, we define the *operational latency* of an event e, $\lambda_o(e)$, as the period between the point in time when e arrives at an operator instance and the point in time when e is completely processed in all assigned windows in this operator instance.

When, at the time of arrival of e, the operator instance is still busy with processing earlier events, e waits in a queue until its processing can start. This is called *queuing latency* of e, $\lambda_q(e)$. Then, e is processed, which induces the *processing latency* of e, $\lambda_p(e)$, the time from starting to process e until e is processed in all assigned windows. Overall, the operational latency of an event is the sum of its queuing latency and processing latency, i.e., $\lambda_o(e) = \lambda_q(e) + \lambda_p(e)$.

To keep the operator latency bound LB_ω, it is required that

$$\lambda_o(e) \leq LB_\omega - \lambda_s - \lambda_m - 2 * \lambda_c \text{ with a probability of } P_{LB}.$$

2.3.3 Operator Configuration

This thesis focuses on two ways of configuring the operator. First, the number of operator instances can be changed dynamically, i.e., the framework is *elastic*. This way, it can react to changing load. The problem how to set and adapt the number of operator instances is addressed in Chapter 3.

Second, the splitter can employ different *scheduling algorithms* in order to assign windows to operator instances. By assigning multiple subsequent overlapping windows to the same operator instance, the communication overhead between splitter and operator instances can be reduced, but at the same time, the operator instance might suffer overload conditions that increase its operational latency. The problem how to control this trade-off is addressed in Chapter 4.

3

Stream Partitioning and Adaptation

In the previous chapters, we have seen that operator parallelization is important in order to handle high workloads in CEP systems. Further, we have noticed that window-based operators are an important building block of CEP systems. Finally, we have outlined that workloads of CEP operators are often heavily fluctuating, while cloud computing allows to dynamically add and remove computing nodes from the CEP system. These observations lead to two research questions: (1) How to split the streams to allow for data parallelization of window-based operators? (2) How to adapt the operator parallelization degree to fluctuating workload, such that low-latency event detection is ensured while cost is minimized? Those research questions are addressed in this chapter as outlined in the following.

First, the problem of consistent and expressive stream partitioning of window-based CEP operators is tackled. As pointed out in Section 1.1.3, the existing stream partitioning methods, key-based [ZR11, Hir12, FMKP13, Ged14, MMTR16, MMA17], batch-based [BDWT13], and pane-based [BT11, KWF+16] partitioning, do not support all window-based CEP operators. To support those operators, the solution presented in this chapter allows for a programmatic description of the window policy by means of a simple API in the splitter. Thus, by exposing the window policies of the operator to the data parallelization framework, expressive parallelization can be supported.

Second, the dynamic adaptation of the resources used for window-based CEP operators to fluctuating workloads is tackled. The goal is that low-latency event detection can be ensured at minimal cost. Latency in event detection can have several causes, such as latency imposed by communication, processing, and queuing of events. While

for communication and processing latencies, bounds can be found that can be met with high probability, unlimited queuing latency can occur when an operator is overloaded and therefore needs to buffer an unlimited amount of events. This can have drastic consequences and dramatically reduces the benefits an IoT application can draw from a CEP system. Therefore, it is of critical importance to develop CEP systems that can guarantee a buffer limit even under high and fluctuating workloads. When facing fluctuating workloads, the parallelization degree must be continuously adapted in order to keep a buffering limit at low resource cost. However, since fluctuations of the workload can have a delayed effect on the imposed processing load in event detection, reactive approaches [FMKP13, GSHW14], i.e., approaches that solely react to changes in the utilization of resources, are not able to ensure a predictable buffering limit. As later also confirmed by our evaluation results, those reactive approaches exceed the acceptable buffering limit at times by a factor of more than 1000.

In this chapter, material published in [MKR14] and [MKR15] is presented. The contributions are threefold: (i) 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. (ii) 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 predictably. (iii) 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 the adaptation methods enforce even under heavily fluctuating workloads a stable parallelization degree and this way ensure that the buffering limit is met and only little over-provisioning of resources is required.

The chapter is structured as follows. In Section 3.1, extensions to the system model presented in Chapter 2 are introduced. Section 3.2 describes the tackled problems and formalizes the guarantees that the system shall provide. The novel stream partitioning model is described in Section 3.3. In Section 3.4, the problem of adapting the operator parallelization degree is tackled. In Section 3.5, an extensive evaluation of the system is presented, showing the performance and efficiency in the context of realistic scenarios in the field of traffic monitoring. Related work is discussed in Section 3.6. Finally, the chapter is concluded in Section 3.7.

3.1 System Model

Extensions of the System Model. The system model in this chapter extends the generic system model described in Chapter 2 as follows. The data parallelization framework is deployed on an infrastructure that consists of a number of computing nodes that are considered failure-free and provide a homogeneous computing capability, i.e., the same CPU and memory capabilities[1]. The number of nodes that can be used by the data parallelization framework is flexible and a sufficient number of nodes is available. This way, new nodes can be allocated for the deployment of operator instances as well as deallocated when they are not used any more. The allocation of a new node and deployment of an operator instance takes *TH (Time Horizon)* time units from the allocation request until the instance is available. The nodes are connected by communication links which guarantee eventual in-order delivery of data.

Extensions of the Data Parallelization Framework. The model of the data parallelization framework in this chapter extends the generic model described in Chapter 2 as follows. Events arriving on the incoming streams of the splitter are stored in-order in a queue. The instances process the events of the assigned windows 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 discarded from the splitter queue. This way, the splitter queue grows or shrinks depending on the ratio of completely processed events to newly arriving events. In particular, the splitter's queue represents a "watermark" of global processing progress of all operator instances; the splitter queue is at least as long as the longest queue of any operator instance. Introducing the splitter queue simplifies the modeling of the adaptation problem in Queuing Theory, as shown later.

Note, that in this chapter, we assume that operators follow the "no consumption" policy, i.e., events can be (re-)used in multiple correlations in multiple overlapping windows. Selection policies and window policies may be arbitrary.

[1] To work with heterogeneous nodes, the methods developed in this chapter can be extended to take into account individual computational capabilities of nodes by employing operator profiles for all available node types (cf. Section 3.4.2). However, homogeneous capabilities are a common case in cloud computing, so that the assumption is practical to focus on the main challenges that are tackled in this chapter.

3.2 Problem Description

As motivated in Chapter 1, timeliness of situation detection is a crucial requirement on a CEP system. That means, that a (probabilistic) latency bound between the occurrence of a situation in the surrounding world and its detection by the CEP system must be kept. In Chapter 2, we have defined the latency bound on a single operator and refined the composition of the operator latency. In particular, we assume that fixed bounds on the latency induced for splitter, merger and communication between the operator components are given. This means that the portion of the overall operator latency that is not fixed is the *operational latency* λ_o, i.e., the latency induced in the operator instances, consisting of the queuing latency λ_q and the processing latency λ_p.

In order to keep the latency bound of an operator, it is crucial that the queuing latency in operator instances is limited. This can only be achieved if the operator throughput keeps up with the incoming event rates. If the operator throughput is permanently lower than the incoming event rate, events are queuing up, leading to unbounded queuing latency. On the other hand, if the operator throughput is permanently higher than the incoming event rate, no queuing of events happens, so that the queuing latency is zero. However, to achieve that, costly overprovisioning of resources is needed, because event rates often fluctuate faster than the operator configuration can react, so that the configuration must be set for worst-case event rates.

The idea of the adaptation mechanism developed in this chapter is to allow for a limited amount of operator overload, i.e., to allow for a limited amount of queuing happening in the operator. This promises to be more cost-efficient, as the resource provisioning does not need to be done in the most pessimistic manner. In this regard, we consider the incoming event rates and the event processing rates of operator instances as probabilistic processes and use Queuing Theory to give probabilistic guarantees on the queuing occurring in the operator instances.

When defining the "amount of queuing" that happens in an operator instance, the first and intuitive definition would be the queuing latency λ_q. However, it has turned out that it is hard to compute probabilistic distributions of λ_q using Queuing Theory. Instead, the *queue length*, i.e., the number of events queued in an operator instance, is more convenient to be used in Queuing Theory. In particular, for some distributions of event arrival rate and event processing rate, *stationary* distributions of the queue length can be computed, yielding the desired probabilistic guarantees on the queue length. From queue length bounds, it is a simple step to deduce queuing latency bounds when event

processing times are known. Note, that this requires that each event in the queue has the same distribution of event processing latencies, i.e., there are no different event processing latencies for different event types, as discussed in Chapter 4. There are two ways to overcome this limitation. A simple, pessimistic way is to perform a worst-case analysis, assuming that all events in the queue are of the event type that has the highest processing latency. A more complex way is to compute the expected processing latency of n events in the queue by taking into account the expected distribution of different event types in the incoming event streams. Sometimes, there is a third way. When an event type does not significantly contribute to the processing load in the operator, it can simply be ignored in the queuing model.

Problem Formalization. Under the setup of Section 3.1, for an 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 constantly adapted so that a user-defined buffering level 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. In 5 % of the cases, the buffer limit may be exceeded, i.e., there are more than 100 events queued at the splitter. This can lead to temporary high latency. By setting the value of $P_{required}$, the users can define how strict their queuing bound, and thus, their latency bound, is.

Solving this problem involves solving two sub-problems: Stream partitioning that leads to correct processing results and the adaptation of the parallelization degree to fluctuating workloads. 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 into the operator logic. To increase the processing rate of an operator, stream partitioning in the splitter should happen independently of processing the partitions in the operator instances. Furthermore, the operator instances should process the assigned partitions while sharing only minimal state amongst each other and needing only minimal synchronization effort. The method for adapting the parallelization degree has to take into account the time needed to deploy a new operator instance on a new computing node.

3.3 Window-based 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 develop a *window-based partitioning model* that is based on the operator model introduced in Chapter 2. The partitioning model is capable of partitioning the incoming event streams of an operator in such a way that the operator instances produce consistent outgoing event streams according to the operator's window policy.

3.3.1 Partitioning Model

Recall that an operator is executed according to its *correlation function* f_ω, mapping windows of events from the incoming streams to detected events that are emitted on the outgoing streams. As the correlation function is a mapping, there is no computational state maintained between two different correlation steps. That means that two windows can be processed on different nodes independently of each other.

The idea of the proposed partitioning model is to split the incoming event stream *by windows* which contain the patterns to be detected, so that we refer to the model as *window-based stream partitioning*. That means that each partition must comprise one or more complete windows, i.e., all events that are part of the window(s).

To ensure that a window is completely contained in a partition, all events between the first and the last event of the window must be part of that partition. Thus, to partition the incoming event streams I_ω, the points where windows 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 window is opened. (ii) The event is part of open windows. (iii) The event triggers that one or more open windows are closed. To evaluate which condition is true, the splitter offers an interface that can be programmed according to the operator's window policy and that provides the ability to store variables that capture internal state, e.g., about window start times. The interface comprises two predicates:

$P_o : e \rightarrow \text{BOOL}$, and

$P_c : (w_{open}, e) \rightarrow \text{BOOL}$.

For each incoming event e, P_o is evaluated to determine whether e opens a window, and P_c is evaluated with each open window w_{open} to determine whether e closes w_{open}.

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

2:  bool Po (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, Window w) begin
11:   if e.timestamp > (w.startTime + windowScope) then
12:       return TRUE
13:   else
14:       return FALSE
15:   end if
16: end function
```

Figure 3.1: Predicates for the time sliding window operator.

Depending on the order of the evaluations of P_o and P_c on a newly arrived event, different semantics can be realized with respect to whether the opening and closing events are part of the window or not.

Examples: In Figure 3.1, we present the predicates for a time-based sliding window operator [ABW06]. Partitions are spanned over all events within a time window that moves by a sliding parameter for each new window. In Figure 3.2, the predicates for a sequence operator Sequence(A;B) [CM94] of two events of type A and B are listed. Partitions are spanned between an event of type A and the next event of type B following it.

3.3.2 Runtime Environment

The runtime environment (RE) manages the execution of an operator instance. It receives the windows assigned by the splitter and enforces that exactly those correlation steps are executed that correspond to the assigned windows. In particular, the RE enforces an isolation between the different windows, i.e., prevents operator instances from processing windows that have not been assigned to them. This is done in the following way: When the operator instance processes an event e that potentially

```
1:  bool Po (Event e) begin
2:   if e.type == "A"  then
3:       return TRUE
4:   else
5:       return FALSE
6:   end if
7:  end function

8:  bool Pc (Event e, Window w) begin
9:   if e.type == "B" AND e.timestamp > w.startTime then
10:      return TRUE
11:  else
12:      return FALSE
13:  end if
14: end function
```

Figure 3.2: Predicates for the sequence operator SEQ(A;B).

opens a new window according to its internal window policy, a function in the RE `isAssigned(e)` is called that signals whether the window starting with e has been assigned to this operator instance or not. Finally, when an event has been processed in all assigned windows it is part of, the RE sends an acknowledgment to the splitter.

3.3.3 Expressiveness

To analyze the window policies for which consistent stream partitioning is supported by the proposed partitioning method, we analyze different categories of CEP operators from the literature. The following categories are considered: Sliding window operators (time-based (Ti) and tuple-based (Tu)) [ABW06], disjunction ($\vee$), sequence (;), conjunction ($\wedge$), aperiodic (A) and periodic (P) operators (all [CM94]) employing an unrestricted consumption mode [AC11], i.e., events can be used in several partitions without restrictions. Figure 3.3 compares window-based stream partitioning, the batch-based partitioning approach proposed in [BDWT13] where event streams are partitioned into batches of a fixed size, and the key-based approaches like [IBY$^+$07, CCA$^+$10, BEH$^+$10, BMK$^+$11, SHGW12, sto14] where event streams are partitioned by a key contained in the events. Batch-based partitioning cannot work consistently with operators for which the maximal amount of events contained in a window depends on the occurring events and, hence, is unknown before run-time, which is the case for (Ti),

	Ti	Tu	∨	;	∧	A	P
Window-based	x	x	x	x	x	x	x
Batch-based	–	x	x	–	x	–	–
Key-based	–	–	x	–	x	–	–

Figure 3.3: Support of consistent partitioning. 'x' denotes supported, '–' denotes not supported.

(;), (A) and (P) operators. Key-based partitioning is even less expressive. It does not allow for the consistent partitioning of event streams based on any other information than keys contained in every single event. Thus, the window context of an event cannot be considered as it would be needed in (Ti), (Tu), (;), (A) and (P) operators.

Window-based stream partitioning does not suffer from those limitations. For all analyzed operator types, the splitting logic is expressive enough to specify the window start and end. The predicates of the time-based sliding window (Ti) operator are listed in Figure 3.1. For the tuple-based sliding window (Tu) operator, the predicates are similar: Instead of a time stamp, a counter is used or sequence numbers are compared to the partition start event. Disjunction (∨) and conjunction (∧) predicates check the corresponding conditions for opening and closing partitions, e.g., attribute values of an event. The sequence (;) predicates are listed in Figure 3.2. For aperiodic (A) and periodic (P) operators, arbitrary start and end conditions can be set, which can be composed of other operators. Accordingly, the predicates from the corresponding operators can be utilized to build the predicates of (A) and (P).

3.4 Adapting the Parallelization Degree

In this section, we describe how to automatically adapt the parallelization degree at changing workloads. We aim for always deploying 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 employ *Queuing Theory (QT)* [GSTH11] to deduce a stationary distribution of the splitter queue length for a given parallelization degree. According to QT, we model the workload and processing latency of an operator by probabilistic arrival and service processes of events. Corresponding to the arrival process, the events in I_ω determine the *workload* of ω. They are streamed from ω's predecessors, resulting in a process of *inter-arrival times* of events. Corresponding to the service process, the *processing latency* of ω is described

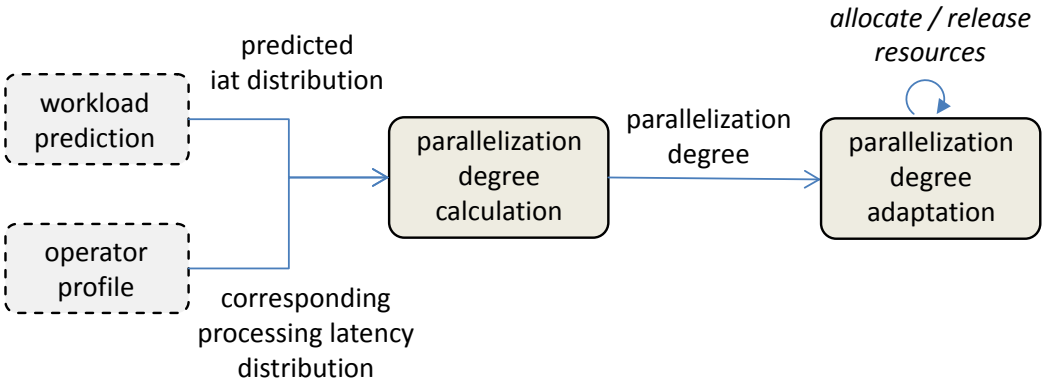


Figure 3.4: Workflow of the adaptation of the parallelization degree.

as the time it takes to process an event completely (i.e., in all windows it is part of). This way, it is possible to configure the parallelization degree solely based on the observed arrival and service processes without interference of the internal operator logic. Fig. 3.4 depicts the interplay of methods in our approach: Models of the predicted future workload and of the corresponding processing latencies are used for continuously calculating the optimal parallelization degree with QT methods. The calculated degree is established by an adaptation algorithm that allocates and deallocates operator instances.

3.4.1 Workload Monitoring and Prediction

A typical workload may consist of three parts [HHKA14]: 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 3.5) and denote t_n the time slice that ends at a point in time n. To this end, we employ 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, can change over subsequent time slices. In Figure 3.5, a schematic distribution of inter-arrival times X in a time slice t_n is depicted as an example. Note that depending on the distribution, all parameters of the distribution can change over subsequent time slices, e.g., the mean and the variance in a normal distribution. To enable workload monitoring, the splitter logs the inter-arrival times of events.

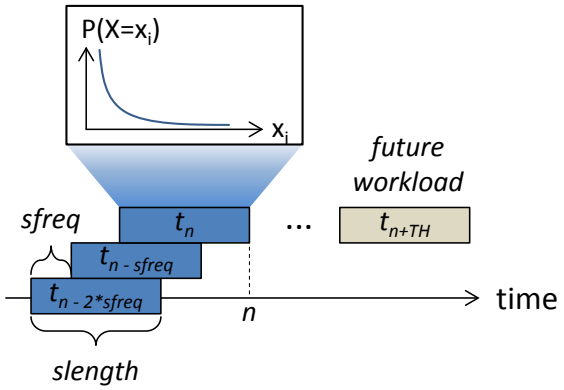


Figure 3.5: Schematic: workload distribution at an operator in a time slice.

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 [Fei15]: The algorithm iterators over a set of supported distributions (exponential, deterministic, normal, uniform, Pareto, log-normal and Weibull 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 the logged data best is determined.

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* [Fei15] is applied. This method is chosen when no higher moments are needed, i.e., when the selected distribution is the exponential, deterministic, normal, uniform or log-normal distribution. Otherwise, the *maximum likelihood method* [Fei15] is applied. With this method, the parameters are calculated such that they would, when sampled from the selected distribution, lead to the given monitored data with the highest probability.

To check the goodness of the selected distribution, statistical tests are applied [Ste74, Fei15]. An important kind of goodness-of-fit test are tests that utilize statistics based on the *empirical distribution function (EDF)* [Ste74]. 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 employs the χ^2 *test for homogeneity*, where random samples from the selected distribution are created and checked against the monitored data [FHC12]. 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 use methods from time series analysis on the workload monitored in the latest time slices. At highly dynamic workloads like Wikipedia page requests, state of the art forecasting methods yield a mean relative error of about 20 % [HHKA14]. Such predictions errors have to be taken into account by adding an overestimation factor when the workload shows characteristics that make it hard to yield accurate predictions. 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.

3.4.2 Operator Profiling

The processing latency of an operator instance depends on the workload. For example, in an operator that employs a time-based window, higher event rates mean that the processed windows contain more events. This can lead to more state gathered in the single windows, such that the processing latency of events in the window increases, and to a higher overlap between subsequent windows, so that more events are processed in multiple windows, which also increases the event processing latency. To account for those dependencies, an operator ω is profiled before run-time for different workloads on the target infrastructure. To create an *operator profile*, first of all an operator instance is instrumented. That means that a monitoring function is added that measures how long it takes to process an event. The operator instance is then executed with different workload distributions that are expected to occur. From the measurements, the corresponding distributions of processing latencies are built with the same probability distribution fitting methods that we use in workload monitoring.

3.4.3 Degree Calculation

Recall that the optimal parallelization degree needs to be recalculated whenever the predicted workload changes. Here, we present the algorithm for the parallelization degree calculation, which is listed in Figure 3.6. Given the predicted workload at time slice t_{n+TH} and the operator profile, the algorithm first calculates the probability that the queue length will be less or equal to BL_ω for a fixed parallelization degree c. If the probability is too small, c is increased, until the minimal c is found that yields a probability higher than or equal to $P_{required}$, which is returned by the function.

```
1:  function calculate_degree () 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
```

Figure 3.6: Algorithm for calculating the optimal parallelization degree.

Let Q = {Q(t) : t $\geq$ 0} be the random process of the queue length Q(t) at time t at a fixed parallelization degree, workload distribution and operator profile. In the core, the algorithm employs methods from QT to calculate the queue length probability P(Q(t) $\leq BL_\omega$), i.e., the probability that there are not more than BL_ω events buffered in the queue. Depending on the workload distribution, the corresponding processing latency 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 P(Q(t) $\leq BL_\omega$) can be calculated. Closed-form formulas to calculate P(Q(t) $\leq BL_\omega$) are available for M/M/c [Bos14] and M/D/c [Tij06] queuing systems[2]. The algorithm uses the available mathematical formulas to compute the queue length probabilities P(Q(t) $\leq BL_\omega$) (line 4). The parallelization degree c is adapted accordingly, until the minimal c is found that yields P(Q(t) $\leq BL_\omega$) $\geq P_{required}$ (lines 5 – 14).

[2]We follow Kendall's notation, A/B/c, A = iat distribution, B = processing latency distribution, c = parallelization degree, M = exponential distribution / Markovian process, D = deterministic distribution / constant. The queue capacity is infinite and the queuing discipline is FIFO in all models used in this work. The actual queue capacity of the system is indeed limited by physical constraints which are, however, typically much higher than the required buffer limits, so that they don't have to be considered in the mathematical model.

```
1:  function adapt_degree () begin
2:    new_degree = calculate_degree()
3:    dif = last_degree − new_degree
4:    if dif < 0 then
5:        cancel |dif| instances in TH time units
6:    end if
7:    if dif > 0 then
8:        Try to recall dif cancellations
9:        n = number of successfully recalled cancellations
10:       Deploy dif − n new instances
11:   end if
12:   last_degree = new_degree
13: end function
```

Figure 3.7: Algorithm for adapting the 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 follow an exponential distribution or are deterministic. 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, i..e., it is pessimistic. 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 it is necessary to have a well-fitting workload and service time classification in order to make sure the approximation yields smaller inter-arrival times and higher processing latencies than the real distributions.

3.4.4 Adaptation

Adaptation of the *parallelization degree*. The algorithm for adapting the parallelization degree is listed in Figure 3.7. It is initiated every *sfreq* time units. The new optimal parallelization degree is calculated with the algorithm from Section 3.4.3 based on the predicted workload in *TH* time units. If there is a difference *dif* between the old and the new degree, the adaptation of the degree in *TH* time units is prepared. There are two cases: (i) $dif < 0$ and (ii) $dif > 0$. In case (i) (line 4), the cancellation of $|dif|$ instances in *TH* time units is registered (line 5). At the time of cancellation, the operator instances are unregistered in the splitter, so that they do not receive new windows. The

instances finish processing the assigned windows and then are shut down. In case (ii) (line 7), 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 (lines 8–9). The rest of the $dif - n$ instances are newly deployed and registered in the splitter as soon as they are live (line 10).

Adaptation of *slength* and *sfreq*. *slength* must be long enough to capture a sufficient number of events to determine the workload distribution with high confidence. For instance, consider 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 [Gue12], *slength* is chosen such that the required number of measurements $num\_measurements$ is 1600, while for $\delta = \pm 10\%$ a value of $num\_measurements = 400$ is sufficient. There is a trade-off between the time needed to collect enough measurements and the statistical significance of the derived workload distribution. If the number of measurements in a time slice is too low, the confidence of the workload distribution and thereby also the workload prediction are affected negatively. However, it is evident that at a low arrival rate of events, it takes a long time until enough measurements are available to reason about the distribution with high confidence. At an average arrival rate of 0.5 events per second, it takes 3200 seconds, that is more than 53 minutes, in order to collect 1600 measurements needed to achieve the aforementioned confidence level.

We propose the algorithm *QT-DYN* that dynamically adapts *slength*: A time slice ends at a maximal time limit $max(slength)$ or when *num_measurements* measurements are available. For instance, we can set $max(slength) = 3$ minutes and $num\_measurements = 400$. In that case, a time slice ends either after 3 minutes or when 400 measurements are taken. This algorithm allows for setting bounds in the trade-off between time and accuracy of capturing the workload distribution. The algorithm is embedded into the procedure of processing measurements (cf. Figure 3.8), checking the conditions when a measurement in a time slice is being processed. The function signals when the time slice is completed, so that the workload distribution can be computed.

In setting the system parameter *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. Another important factor is how fast the workload is expected to change and what is the impact on the parallelization degree. If it changes often and very abruptly and a small change has high impact on the parallelization degree, a lower value of *sfreq* can be beneficial.

```
1: function process_measurement (time_slice t, measurement m) begin
2:   if (m.timestamp ≥ t.start + max(slength))
3:       or (t.#measurements ≥ num_measurements) then
4:       t.COMPLETED( )
5:   else
6:       t.ADD_MEASUREMENT(m)
7:   end if
8: end function
```

Figure 3.8: Processing measurements in time slices with the QT-DYN algorithm.

3.5 Evaluation

In this section, we evaluate the proposed method to adapt the operator parallelization degree by analyzing the queue lengths and processing latencies at different workloads. We compare the QT-based approach to a reactive approach that adapts the parallelization degree based on the average CPU utilization (cf. Section 3.5.2). 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.

3.5.1 System Parameters

All experiments were performed on a computing cluster consisting of 6 physical hosts with 8 CPU 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 typical boot times, e.g., in the Amazon EC2 Cloud[3].

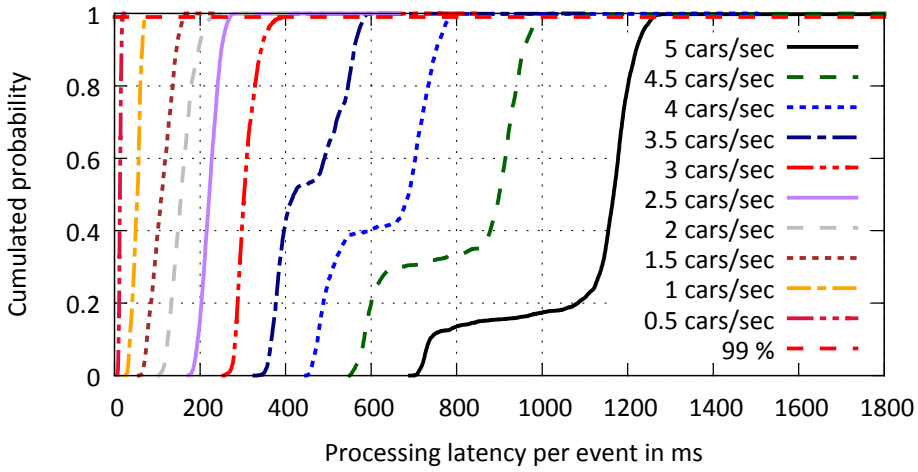


Figure 3.9: Operator profile of $\omega_{overtake}$.

3.5.2 Dynamic Degree Adaptation

Traffic Monitoring Scenario

Recall the traffic monitoring scenario introduced in Section 1.1.2. Given two sensors with synchronized clocks deployed at the beginning and end of a no-passing zone (*L1* and *L2*), the operator $\omega_{overtake}$ detects when a vehicle overtakes another one. We assume the following parameters of the setup: Between *L1* and *L2*, there is a distance of 15 kilometers. Following the speed limit, the vehicles in average drive 60 kilometers per hour; however, there is a ratio of 10 percent 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 follows a Poisson process [MM13], thus resulting in exponential distribution of inter-arrival times of vehicles at the checkpoints.

Operator Profile

Fig. 3.9 shows the measured operator profile. For sake of simplicity, we only consider events of type *L2*, because in the implementation of $\omega_{overtake}$, events of type *L1* are just added to a list which is a negligible operation while events of type *L2* are compared to all events of type *L1* that have occurred in the window, leading to noticeable processing latency. The processing latency grows polynomially with the event rate. For employing

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

QT, we approximate the profile with the deterministic values of the 99th 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 queuing latency of less than 4 seconds even at the highest traffic density. This allows for direct feedback to a driver who violated the traffic rules. In our experiments, we choose $sfreq = slength$, i.e., subsequent monitoring windows are non-overlapping.

Fixed Average Inter-arrival Time

Figure 3.10a shows the measured queue sizes of our QT-based approach, with a fixed *slength* at 1600 events, compared to a reactive approach that adapts the parallelization degree depending on the average CPU load of active operator instances. The reactive approach adds a new instance once the average CPU load over two subsequent time frames of 5 seconds is higher than 70 % and removes one instance when it is less than 50 %. Such reactive approaches have been used in related work; here, we follow the settings of a reactive controller used by Fernandez et al. [FMKP13] in their SEEP stream processing system. In the QT approach, we used naive forecast [HHKA14], 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 each 10 seconds over an experiment run time of more than 8 hours. As can be seen in Figure 3.10a, 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 reactive CPU-based approach fails to keep BL_ω; the 95th percentile is at more than 17,000 events.

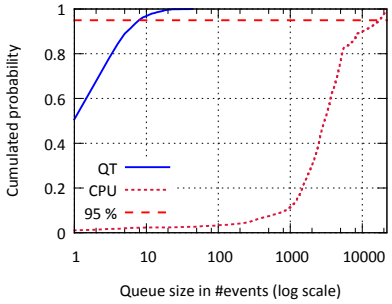
Furthermore, we measured the parallelization degree that is deployed in the different adaptation approaches (Figure 3.10b). In the QT-based approach, a stable degree of 8 is kept, as the workload classification reliably detects the workload distribution. The reactive CPU-based approach, however, does not stabilize the degree. Instead, it is gradually increased to a certain level and then suddenly drops, taking another 30 minutes to increase again. The reason is that the traffic monitoring operator works on large, overlapping windows. The processing effort per event depends on the number of windows that contain the event as well as the number of events that already have been processed within a window, as a new event from *L2* needs to be compared to a growing number of events from *L1* in a growing window. Initially, the reactive CPU-based approach assigns many windows to a small number of instances, because the

CPU load in the beginning of processing a window is still small. When the CPU load grows, it is already too late to add new instances, because the windows that cause the overload are already assigned to the small number of instances. Note, that we do not migrate intermediate window states in our approach, so that the effects of overloading an operator instance with too many windows lasts until the windows are closed again. The algorithm adds more and more instances, and finally overshoots. Then, the same cycle starts again.

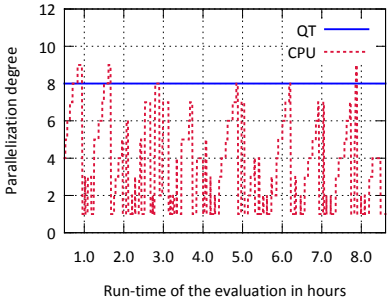
Dynamic Average Inter-arrival Time

In the second scenario, the average inter-arrival time (iat) 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 strategies for choosing *slength*: (i) Fixed sizes of the time slices containing $num\_measurements$ measurements (denoted by QT-$num\_measurements$), and (ii) the bounded dynamic algorithm QT-DYN (cf. Section 3.4.4) with $max(slength)$ = 3 minutes and $num\_measurements = 400$.

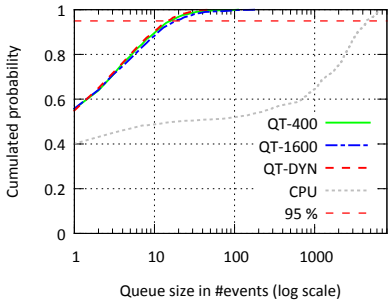
Figure 3.10c shows the cumulative distribution function (cdf) of the queue sizes for the QT-400, the QT-1600, the QT-DYN, and the reactive CPU-based approach, where the 95-percentile is at 15, 18, 14 and 4,500 respectively. Looking into the logs of the QT-1600 approach, higher queue sizes than 15 mostly happen while scaling up, as it takes some time to gather 1,600 iat measurements. For instance, gathering 1,600 iat measurements at an event arrival rate of 4 events per second (corresponds to 4 cars per second passing L_2 in our traffic scenario) would take 400 seconds. Using a smaller number of measurements in QT-400 and QT-DYN diminishes the problem while still giving a good estimation of the workload distribution. In Figure 3.10d, the parallelization degree of the QT and reactive CPU-based approach 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 reactive CPU-based approach. The dynamic algorithm QT-DYN shows a good adaptation behavior, as it allows for fast adaptations at a low event rate while increasing the confidence at a high event rate.



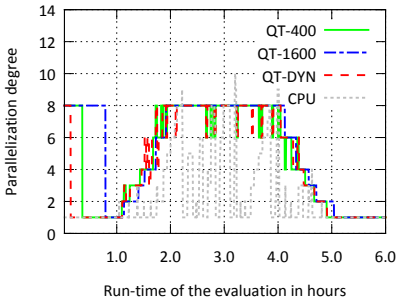
(a) Queue sizes of the QT and CPU approaches at static average iat.



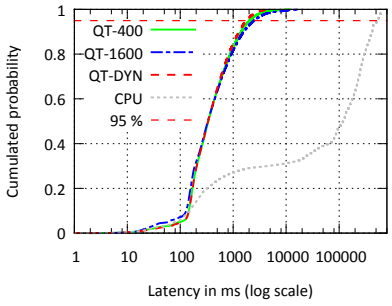
(b) Parallelization degree of the QT and CPU approaches at static average iat.



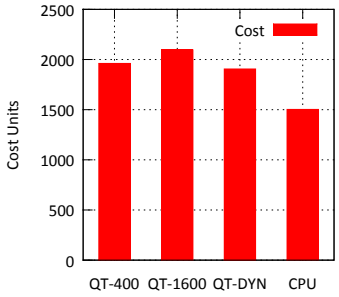
(c) Queue sizes of the QT and CPU approaches at dynamic average iat.



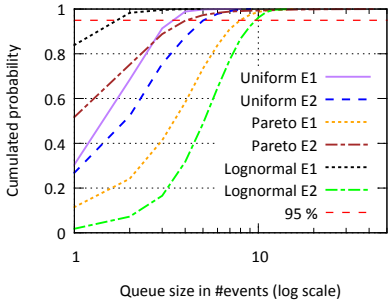
(d) Parallelization degree of the QT and CPU approaches at dynamic average iat.



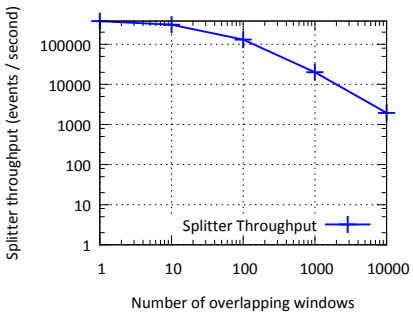
(e) Latency in the QT and CPU approaches at dynamic average iat.



(f) Cost induced by the QT and CPU approaches at dynamic average iat.



(g) Queue sizes at other distributions that are approximated.



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

Figure 3.10: Evaluation results for different scenarios.

To further elaborate on the impact of the parallelization degree, we analyzed the imposed cost when running the dynamic scenario with the QT algorithms and the reactive CPU-based approach. We measure the cost for running operator instances based on a low pricing granularity of 1 minute, like it is given, for instance, in Google's cloud computing offerings[4]. For the sake of generality, we assume that occupying a computing node that can host one operator instance for one minute costs 1 Cost Unit (CU). Figure 3.10f shows that in the QT approach, QT-DYN performs best with 1906 CU, followed by QT-400 with 1961 CU (+ 2.9 %) and QT-1600 with 2096 CU (+ 7.1 %). Furthermore, the reactive CPU-based approach yields the lowest cost with 1502 CU. However, it fails to meet the buffer limit.

To elaborate on the relation between the queue length and the total operator latency, we have measured the *end-to-end latency* of the operator. That is the period between the point in time when an event that ends the detection of a pattern arrives at the splitter and the point in time when all corresponding outgoing events have been created. Figure 3.10e shows that the reactive CPU-based approach imposes a much higher latency than the QT-based approach. In 5 % of the cases, the latency is even higher than 530 seconds and only in 29 % of the time a low latency of under 2 seconds is achieved. In contrast to this, the QT based approach yields a 95th percentile in latency of 1759 ms with QT-DYN, 2036 ms with QT-400 (+ 15.7 %), and 2471 ms with QT-1600 (+ 40.5 %).

Analyzing the results, it is evident that QT-DYN yields a better overall performance than QT-400 and QT-1600. While the improvement of the buffering level in comparison to QT-400 might not be significant and could be interpreted as a statistical deviation, the improvements in cost and especially in latency are clear.

3.5.3 Approximating Distributions

Recall that according to the considerations in Section 3.4.3, iat and processing latency distributions that do not yield a stationary queue length distribution in a QT queuing system are approximated by exponential or deterministic distributions. To analyze the effects of such approximations, we consider several distributions: (i) The Pareto distribution representing *heavy tailed, skewed* distributions, (ii) the uniform distribution representing *short tailed* distributions, and (iii) the log-normal distribution representing the multiplicative *product* of many independent, positive random variables. We run two experiments for each distribution:

[4] https://cloud.google.com/compute/pricing

	Parameters	E1	λ^{-1}	c	E2	D	c
Uniform	a = 100, b = 200		0.0432	**10**		0.199	**4**
Pareto	$x_{min} = 0.05$, $k = 2$		0.0667	**6**		0.5	**10**
Log-normal	$\mu = -1.0$, $\sigma = 0.2$		0.1266	**3**		0.586	**12**

Figure 3.11: Parameters, corresponding approximations and calculated parallelization degrees for the approximated distributions.

E1: In the first experiment, the iat follows the approximated distribution while the processing latency follows an exponential distribution with an average value of 300 ms. In the experiment, the given iat is approximated by an exponential distribution, building an M/M/c queuing system.

E2: In the second experiment, the iat follows an exponential distribution with an average value of 66.7 ms while the processing latency follows the approximated distribution. In the experiment, the given processing latency is approximated by a deterministic distribution, building an M/D/c queuing system.

We have chosen different parameters for the distributions, which are all listed in Figure 3.11 in the column "Parameters". In the E1 and E2 section in that table, the parameters of the corresponding approximations are listed, i.e. the average value "λ^{-1}" of the exponential approximation of the iat in E1 and the value "D" of the deterministic approximation of the processing latency in E2. In both experiments, "c" denotes the parallelization degree computed with QT formulas.

BL_ω is kept in all experiments (cf. Figure 3.10g), so that the approximations show good results.

3.5.4 Splitter Throughput

In the splitter, the predicate logic for typical operators consists of only a fixed number of parameter comparisons that are performed for each event in P_o and P_c. What scales

up the effort in evaluating the predicates is the number of concurrently open windows, as for each open window, 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 3.10h shows that the splitter yields high throughput that degrades proportionally to the number of overlapping windows.

3.6 Related Work

Besides window-based splitting, as proposed in this chapter, other splitting approaches in data parallel stream processing are key-based, batch-based and pane-based splitting. Those approaches are discussed in Section 1.1.3.

In case of operator overload, two situations can occur: (i) The operator applies *load shedding*, i.e., discarding events that it is not capable of processing in time [TcZ07, CJ09, KCFP12]. This apparently leads to inconsistencies (false-negatives or false-positives), which are not tolerable in many scenarios. (ii) A high amount of events need to be buffered before being processed, which can cause an unacceptable latency in event detection. To avoid this case, the throughput of the operator has to be adapted to the workload. Different approaches have been proposed to dynamically adapt the operator parallelization degree.

In terms of elasticity approaches, we can differentiate between *reactive* approaches and *proactive* approaches. Reactive approaches decide about scaling based on live-feedback from the CEP system. Examples for feedback parameters are CPU load, measured latency or throughput of operator instances. In contrast to that, proactive approaches predict the future workload and resulting system behavior beforehand, so that adaptations of the parallelization degree can be issued earlier. In the following, we discuss reactive and proactive approaches from literature in terms of the method they use for deciding when to adapt the operator parallelization degree and the guarantees that the elasticity control provides in terms of timeliness of event processing.

Schneider et al. [SAG+09] propose a data parallelization framework for stateless elastic operators on multi-core hosts. The thread level, i.e., number of threads active in an operator, is increased or decreased in a greedy manner based on the stability of the operator throughput, until a stability condition is met and the thread level is kept. The system does not provide any guarantees on throughput or latency. The reactive scale-out approach proposed by Fernandez et al. [FMKP13] scales the op-

erator 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. Similarly, the elasticity mechanism in StreamCloud by Gulisano et al. [GJPPnM+12] as well as MillWheel by Akidau et al. [ABB+13] are based on a threshold on the average CPU utilization in the system and cannot provide guarantees on latency. In [GSHW14], a more sophisticated elasticity algorithm is proposed that uses the feedback parameters congestion and throughput. However, it does not guarantee buffer limits at bursty workloads. Heinze et al., based on their FUGU system [HJP+13], analyze three different auto-scaling strategies [HPJF14]: Global threshold-based, local threshold-based, and model-based with reinforcement learning. However, their approach does not guarantee latency bounds. Later, Heinze et al. [HJHF14] propose a hybrid reactive and model-based controller for elastically scaling the number of machines used in order to deploy the operator graphs of multiple queries. The goal of their approach is to keep an average end-to-end latency bound. Based on that work, Heinze et al. in their later work [HRM+15] propose an online parameter optimization method for automatically tuning six different thresholds of reactive scaling methods that scale in or out the number of virtual machines that host a set of operators. As their work addresses key-based data parallelization using state migration when changing the parallelization degree of an operator, it cannot be directly applied to window-based data parallelization without state migration, as proposed in our work. Mencagli [Men16] proposes a reactive game-theoretic *distributed* controller for elastic stream processing operators. The operators implement a strategy to achieve a Nash equilibrium in a game, where operators increase their parallelization degree when they detect that they are a system bottleneck. Mencagli shows that a cooperative strategy, where incentives are provided to promote cooperation between different operators, bring the system closer to the global optimum than a purely selfish strategy. The game settings aim to maximize the system throughput without giving any specific guarantees on latency.

The proactive approach of Balkesen et al. [BTO13] tries to forecast the exact event arrival rate and assumes a fixed per-tuple processing latency when determining the optimal parallelization degree, which does not always hold, as we show with the operator profiles in our traffic monitoring scenario (cf. Section 3.5). De Matteis and Mencagli [DMM16] propose latency-aware and energy-efficient scaling of key-based data parallel stream processing operators on multi-core machines based on a Model Predictive Control strategy. Similar to the our work, a disturbance forecaster predicts the future event arrival rate and processing latency. To predict the average per-tuple

latency, a formula from QT is applied (Kingman's formula), which depends on the system utilization and the coefficients of variation of the inter-arrival time and the service time. Contrasting to our work, this makes the model more widely applicable, as it works with G/G/x queue models (i.e., no particular distributions are assumed), but the model predicts average latency instead of queue length distributions, and hence, does not allow for providing worst-case guarantees. In [DMM17a], De Matteis and Mencagli extend their approach to horizontal scaling across several computing nodes, building on the same predictive control model. Lohrmann et al. [LJK15] propose a controller that adapts the parallelization degree of all operators in a distributed operator graph such that an average latency bound is met. The controller employs a QT model that is based on Kingman's formula, similarly as in De Matteis and Mencagli [DMM16], to predict the queuing latency in the system. A reactive component doubles the parallelization degree if an operator, against the predictions of the QT model, becomes a bottleneck, so that bottlenecks are quickly removed.

Performance modeling approaches, e.g., Queuing Petri Nets [Kou06] or latency estimation models like Mace [CGB+11], require a deeper knowledge of the operators and do not provide methods to adapt the parallelization degree in order to yield a limited buffering level at fluctuating workloads. QT has also been applied in other related fields to model or predict queuing latency, such as process mining [SWGM15] and business process management [SWG+15].

3.7 Conclusion

In this chapter, we have identified two important shortcomings on the way towards low latency event detection in CEP systems. First, the state of the art lacks consistent parallelization models for a large 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 the window-based CEP operators. Furthermore, the proposed QT-based degree adaptation method is able to meet probabilistic buffering limits at highly fluctuating workloads.

4

Bandwidth-Efficient Scheduling

We have seen in the previous chapters that in a window-based data parallelization framework for CEP operators, incoming event streams of an operator are split into windows that can be processed in parallel by an arbitrary number of operator instances. Those operator instances do not share state, so that they can be hosted on shared nothing hosts, e.g., virtual machines in a cloud data center, which allows for high scalability and elasticity. To ensure consistency, each window contains all events needed in order to detect a pattern. This means that different windows can overlap, i.e., events are part of multiple windows [BDWT13, MKR15]. When splitting incoming event streams, the data parallelization framework assigns a window to an operator instance when the start of the window is detected. In doing so, assigning overlapping windows to different operator instances results in increasing communication overhead, as events that are part of multiple different windows are replicated to multiple operator instances. In the worst case, an event may be transmitted to all operator instances, leading to a high network load. In cloud data centers, this may not only impair the performance of the hosted CEP system, but also the performance of other applications hosted on the same infrastructure. Network-intensive applications have been identified as a major cause of bottlenecks in cloud data centers [GHJ+09, BCKR11, LDGB13]. Therefore, reducing the bandwidth consumption of parallel CEP systems can be of great worth to all hosted applications.

To reduce the bandwidth consumption, we employ batch scheduling of subsequent overlapping windows, i.e., assigning them to the same operator instance. That way, events from the overlap only need to be transferred once. However, at the same time, the operator instance must process more windows in a shorter time. This can lead to

temporary overload, so that events are buffered and queuing latency is accumulating. Nevertheless, the latency between arrival of an event and its successful processing must not exceed a given latency bound. We address the following challenges in batching the optimal amount of windows, which cannot be solved with state-of-the-art scheduling algorithms from stream processing [CcR+03, LMT+05, BT11].

- **Per-event latency**: Each incoming event at an operator can potentially trigger the detection of a pattern leading to a situation detection. Therefore, a latency bound should be kept for *each single event*.
- **Window overlap**: The overlap between windows of a batch influences the processing load induced by each event, as each event is processed in the context of each window it is part of. Moreover, the scheduling decision is made on open windows, i.e., the events and the overlap of a window are not known at scheduling time.
- **Automatic adaptation**: A batch scheduling controller should be able to automatically adapt to changing workload conditions without being manually trained for those conditions beforehand.

In this chapter, material published in [MTR16] and [MTR17] is presented. The following contributions are provided. (1) Based on evaluations from different CEP operators, we identify key factors that influence the latency in operator instances. In particular, we identify factors that have not been regarded in related work before. (2) Taking into account the identified key factors, we propose a model-based batch scheduling controller. The model allows to predict the latency induced in operator instances when assigning windows. (3) We provide extensive evaluations of the system behavior in two different scenarios, showing that our approach minimizes communication overhead while operator instances keep a required latency bound even when the system faces heavily fluctuating workloads.

4.1 Problem Description

The system model assumed in this chapter is the same as in Chapter 3. Further, we assume the data parallelization framework introduced in Chapter 2 with the window-based, pattern-sensitive stream partitioning method introduced in Chapter 3.

In particular, upon detection of the start of a new window, this window is assigned to an operator instance according to a scheduling algorithm. In an operator instance,

incoming events are processed sequentially. Within each window, an event has a different context. Therefore, when processing an event e, the operator instance sequentially processes e in the context of each window that e is part of.

Example: In the scenario in Figure 4.1, the pattern to be detected is "within one minute after occurrence of an event of type A, a sequence of events of type B and C occurs". In the extended `MATCH-RECOGNIZE` notation [ZWC07] introduced in Section 1.1.2, this query can be formalized as follows:

PATTERN (A B C)
DEFINE
A **AS** $A.type = A$
B **AS** $B.type = B$
C **AS** $C.type = C$
WITHIN 1 minute **FROM** $A.timestamp$

The splitter opens a window whenever an event of type A occurs, and closes the window after one minute. The operator instances check whether in a window, events of type B and C occur in the right order. Taking a look at the splitting, we see that all events following A_1 within one minute are part of the same window w_x: If some of the events would be missing, they could not be checked for the Sequence(B;C) sub-pattern that follows A_1. In the example, two overlapping windows w_x and w_y have been assigned to the same operator instance i. When i processes an event, e.g., C_1, this event has a different context in w_x than in w_y: In w_x, Sequence(B;C) is detected, while in w_y, the sequence is not detected. In checking the occurrences of the sequence pattern in different windows, operator instance i processes C_1 sequentially first in w_x and then in w_y.

Because scheduling windows to operator instances significantly influences the latency induced in each operator instance, in this chapter, we focus on batch scheduling suitable amounts of windows to operator instances such that a latency bound in those operator instances is kept.

Problem Formalization: To minimize the communication overhead, the batch scheduling controller tries to assign as many subsequent windows as possible to the same operator instance subject to the constraint that the operational latency of events in that instance must not exceed a latency bound LB. As soon as the start of a new window w_{new} is detected by the splitter, the batch scheduling controller decides whether

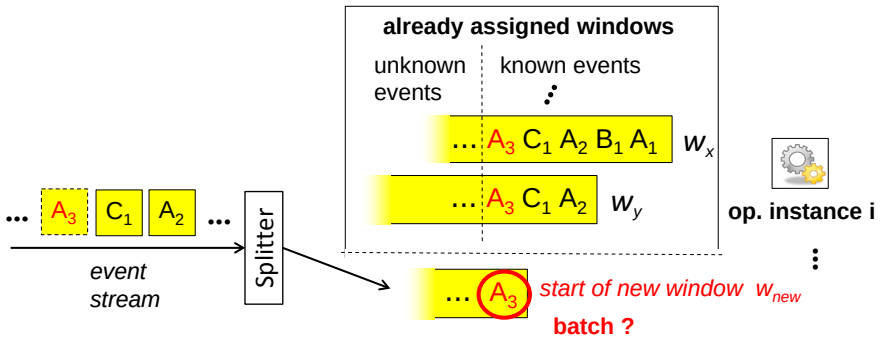


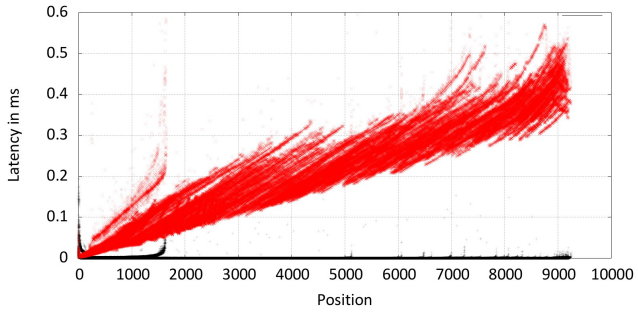
Figure 4.1: Splitting and scheduling.

assigning that window to the same operator instance as the previous window would cause operational latency of events to exceed *LB*. The batch scheduling controller does not know in advance all the events in w_{new}, which makes the decision very challenging. We denote this problem as the *batch scheduling problem* in data-parallel CEP operators.

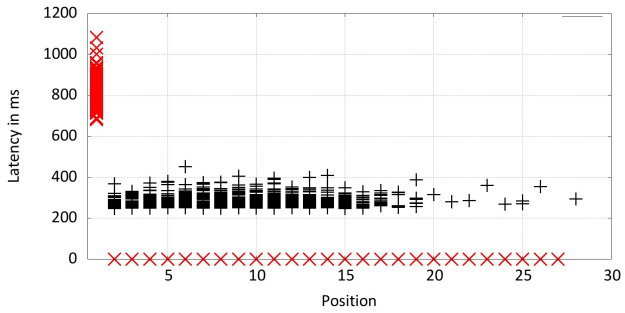
The trade-off tackled in the batch scheduling problem is exemplified in Figure 4.1. An event A_3 arrives at the splitter and the splitter detects that A_3 starts a new window w_{new}, which now has to be scheduled. Let us suppose that a set of previous windows W_{old} =(..., w_x, w_y) has already been scheduled to a specific operator instance i. Events before A_3 in W_{old} have been transferred to operator instance i. However, further events arriving after A_3 can as well be part of some of the windows in W_{old}; hence, they are transferred to operator instance i, too. When scheduling w_{new} to operator instance i, communication overhead can be reduced, because events overlapping between w_{new} and W_{old} do not need to be transferred to multiple different operator instances. On the other hand, they need to be processed additionally in the scope of w_{new}, inducing higher processing latency in operator instance i. The splitter has to decide whether w_{new} can be assigned to operator instance i such that the operational latency does not increase beyond *LB*.

4.2 Batch Scheduling

To analyze the batch scheduling problem, in this section, we make the following contributions. First, in Section 4.2.1, we identify and thoroughly analyze *key factors* that influence the operational latency in an operator instance. We conclude that the impact of key factors on operational latency in an operator instance is complex and depends



(a) Traffic monitoring: Processing latency of events in different positions in a window. *L1* events: black, *L2* events: red.



(b) Face recognition: Processing latency of events in different positions in a window. Face events: black, query events: red.

Figure 4.2: Evaluation of processing latency.

on the workload as well as on the operator. Then, in Section 4.2.2, we highlight the difficulties in developing a reactive batch scheduling controller that works without a latency model.

4.2.1 Key Factors

In the following, we first identify and analyze key factors that influence the processing latency of events in the scope of a *single* window. Based on that, we identify and analyze key factors that influence operational latency in a whole *batch* of windows. To this end, we evaluate two different CEP operators: a traffic monitoring and a face recognition operator (cf. Section 1.1.2). We ran all experiments on the computing cluster described in Section 4.4 with a parallelization degree of 8.

Processing latency of events in a window. When processing a *single* window in an operator instance, each event imposes a specific processing latency. This is different from stream processing where the processing latency of an event in a window is con-

sidered fixed [BT11,ZR11]. We identified two key factors that influence the processing latency of an event in a window: its type and its position.

Event type. Event types are a fundamental concept in CEP. Many query languages, such as Snoop [CM94], Amit [AE04], SASE [WDR06] and Tesla [CM10], allow for the definition of event patterns based on event types—e.g. Sequence(A;B), a sequence of events of type A and B. In the traffic monitoring operator, different event types are processed in a different way. *L1* events are simply added to a list of seen events, while *L2* events are compared to the seen events (cf. Figure 4.2a). In the face recognition operator, *query* events are processed by building a face model of the queried person, while *face* events are processed by comparing them to the established face model of the window (cf. Figure 4.2b). In both operators, we see different processing latencies depending on the event types.

Position of event. When processing events of a window, internal state is gathered in an operator [FMKP13, BDWT13], which can influence the processing latency of events. For instance, in the traffic monitoring operator, an *L2* event e_{L2} can potentially complete a sub-pattern Sequence(B;C) with $B.type = L1$ and with $C.plate = B.plate$ and $C.type = L2$ and this way complete the pattern (cf. the query formalization of $\omega_{overtake}$ provided in Section 1.1.2). Therefore, e_{L2} is compared to all *L1* events that have been seen in the window before (*equi-join* operator). Thus, with a higher position of e_{L2}, its processing latency increases, as evaluated in Figure 4.2a. However, the processing latency of events does not necessarily increase with position. In the face recognition operator, each face event is compared to a query event; the `face_match` function imposes the same processing latency in each event position (cf. Figure 4.2b).

Operational latency in a batch of windows. In a *batch* of windows, different windows may overlap. When the batch scheduling controller assigns a window to an operator instance that overlaps with other windows, the processing latency of all events in the overlap is influenced, as events are processed sequentially in the scope of their windows. Recall that a window has to comprise all events needed in order to detect a queried pattern. Therefore, the overlap of different windows cannot be changed by the batch scheduling controller. That is different from batch scheduling problems handled in stream processing, where batches are considered to be arbitrarily large, *non-overlapping* sets of events, and batch scheduling decides how many *events* shall be batched to a processing node [LMT+05, DZSS14].

In the following, we identify key factors influencing the overlap of windows and analyze their impact on operational latency in operator instances. To this end, we run

scenario parameters				measurements			
#	batch size	avg. iat (s)	*ws* (s)	max. op. latency (s)	feedback delay (s)	max. queue length	feedback delay (s)
1	500	0.15	900	2.4	725.5	15	773.6
2	500	0.125	900	3.7	757.7	27	724.8
3	500	0.1	900	24.1	699.0	248	810.2
4	750	0.1	900	100.6	800.8	1029	844.0
5	1,000	0.1	900	116.1	824.3	1194	795.6
6	1,000	0.1	1000	197.8	1,041.8	1699	999.0
7	1,000	0.1	1100	199.2	1,179.2	1898	1,100.0

(a) Traffic monitoring operator.

scenario parameters				measurements			
#	batch size	avg. iat (s)	*ws* (s)	max. op. latency (s)	feedback delay (s)	max. queue length	feedback delay (s)
1	10	0.667	10	37.9	46.3	43	10.1
2	10	0.4	10	68.7	77.1	84	9.4
3	10	0.286	10	99.7	108.0	115	10.6
4	15	0.286	10	145.1	153.2	164	8.3
5	20	0.286	10	195.1	200.7	191	10.2
6	20	0.286	15	289.4	301.8	234	14.4
7	20	0.286	20	392.1	410.1	258	19.6

(b) Face recognition operator.

Figure 4.3: Max. operational latency, queue length and feedback delays.

experiments with the traffic monitoring operator and the face recognition operator. In each experiment, using different traffic densities and different numbers of persons in a video frame, one key factor value is changed while all other key factors are kept constant, and the differences in operational latency peaks are analyzed (cf. Figure 4.3). For each experiment, more than 370,000 measurements have been taken.

Batch size. The batch size, i.e., number of windows assigned to an operator instance in a batch, influences the overlap of the windows, and hence, the operational latency of events. However, the relation between batch size and operational latency peak is not trivial. In the traffic monitoring operator, increasing the batch size by 50 % and then by further 33 % induces an increase in operational latency peak by 317 % and 15 %, respectively (cf. Figure 4.3a, #3, #4 and #5). In the face recognition operator, the relation between batch size and operational latency seems to be proportional (cf. Figure 4.3b). We suppose that this irregular behavior of the operational latency peak originates in the queuing behavior of operator instances. As long as an operator instance is not overloaded by the scheduled batch of windows, queuing latency is moderate, mostly caused by small event bursts in the incoming event stream. When an overload occurs, queuing latency increases very abruptly, so that operational latency peaks increase rapidly (e.g., by 317 % in the traffic monitoring operator from experiment #3 to experiment #4). After that, adding more overload by increasing the batch size even further increases the operational latency peaks more smoothly, i.e., not in an abrupt way.

Inter-arrival time (iat). Given a fixed batch size, the inter-arrival time *iat* of events influences the queuing latency of events. Further, it can influence the number of events in the windows, e.g., in time-based windows. The number of events in windows influences their overlap, which, in turn, influences the processing latency of the events. Thus, there is a complex relation between *iat* and operational latency. In the traffic monitoring operator, we decreased the average *iat* of events first by 17 %, and then by further 20 %. This induced an increase in operational latency peak by 54 % and 551 %, respectively (cf. Figure 4.3a, #1, #2 and #3). Similarly, in the face recognition operator, decreasing the average *iat* of events first by 40 % and then by further 28.5 %, led to an increase in operational latency peak by 81 % and 45 %, respectively (cf. Figure 4.3b).

Window scope (ws). The window scope *ws*—i.e., the time stamp difference between the start and end event of a window—depends on the queried patterns to be detected by the CEP operator. It can be fixed to a specific time, e.g., when the query depends on

a time-based window [ABW06], but it can also depend on the occurrence of specific events, e.g., in aperiodic queries or queries that define a sequence of specific events [CM10, CM94]. For instance, in the traffic monitoring operator, the start and end of a window depend on the speed of the vehicles, as a window starts when a vehicle passes *L1* and ends when the same vehicle passes *L2*. When the speed of a vehicle is lower, the time spanned by the window opened from this vehicle is larger. Therefore, the size and overlap of windows can change even when the batch size and *iat* stay the same. This is different from stream processing, where only windows of fixed size and fixed slide— time- or count-based—are analyzed [BT11]. In the traffic monitoring operator, we increased *ws* by 11 %, and then by further 10 %. This induced an increase in operational latency peak by 70 % and 1 %, respectively (Figure 4.3a, #5, #6 and #7). In the face recognition operator, however, increasing *ws* led to a proportional increase in operational latency peaks.

From the observations on key factors that influence operational latency when processing a batch of windows, we conclude that building a direct mapping from *batch size*, *inter-arrival time* and *window scope* to operational latency peaks in operator instances is hard. The relation between key factors and operational latency peaks that occur in operator instances is complex, and different in different operators. A model trained before run-time (off-line) or at run-time (online), hence, does not suffice; due to the complex relations between key factors, it is hard to train a model that can make reliable predictions outside of the learned parameter value ranges. Further, domain knowledge alone is not enough in order to hand-craft a latency model: Knowledge about the operator implementation does not necessarily help in understanding the relations between the identified key factors and the operational latency peak.

In the following, we discuss whether the need for a latency model predicting the operational latency can be completely avoided by employing a reactive batch scheduling controller.

4.2.2 Reactive Controllers

Here, we discuss the difficulties involved in devising a reactive batch scheduling controller. Reactive controllers are widely used in scheduling algorithms in the related field of parallel stream processing systems [LMT+05, DZSS14]. The basic idea of a reactive controller is that it schedules windows according to *feedback parameters* (like operational latency or queue length) from the operator instances that indicate

how many windows can be batched. In the following, we point out the differences in batch scheduling in data-parallel CEP operators to scheduling problems that have been solved with reactive controllers. Then, we analyze operational latency and queue length of operator instances in the scope of the scenarios used in Section 4.2.1 in detail and show that none of these parameters provides reliable feedback to implement a reactive controller.

In data-parallel CEP operators, in order to maintain the latency bound for each event, the batch scheduling controller decides at the *start* of a window to which instance this window is scheduled. Then, it directs all events that arrive in the scope of that window to the corresponding instance. It is infeasible for the controller to wait until all events of the window are present and then schedule the window; it would take too much time in view of per-event latency bounds. After assigning a window to an operator instance at the occurrence of its start event, many other events of that window arrive until the window is finally closed. Thus, over the *whole time span* of the window, feedback parameters in the operator instance are influenced by the scheduling decision, i.e., a long time after the scheduling decision has been made. That poses a completely different problem from other batch scheduling problems that are tackled with reactive batch scheduling, e.g., the problem of scheduling batches of events in streamed batch processing [DZSS14], where a controller *first* builds a batch of available events and *then* assigns it to an operator instance.

Therefore, in data-parallel CEP operators, there can be a high delay between the assignment of a window to an operator instance and the occurrence of the peak value of the feedback parameters in that operator instance. We denote this delay as the *feedback delay*. In Figure 4.3a, we have measured the feedback delay of operational latency and of queue length in the different runs of the traffic monitoring operator under different conditions; a feedback delay of 699 to 1,179 *seconds* occurred for both parameters. In that time, many subsequent batch scheduling decisions have to be made by the controller. At the same time, key factors like inter-arrival time, window scope, event types, etc. continuously change. Moreover, the feedback delay is not constant, so that the controller cannot rely on it; it is not clear whether the feedback parameter measured in an operator instance is already the peak value or how much further it will grow.

To mitigate high feedback delays, we devise a *latency-reactive* controller that reacts on the *current* operational latency in operator instances. Windows are batched to the same operator instance until at the instance, the current operational latency reaches a threshold TH; subsequent windows are scheduled to the next operator instance. This,

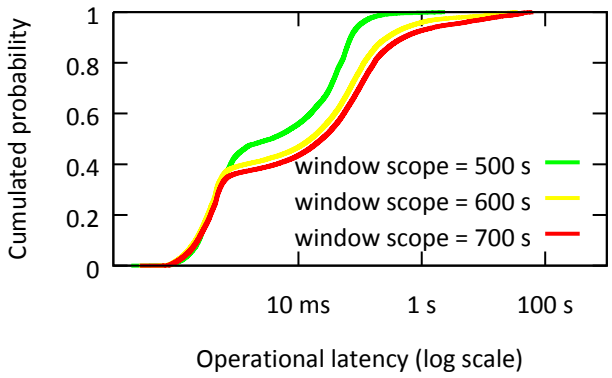


Figure 4.4: Traffic monitoring: Operational latency with reactive batch scheduling at $TH = 100ms$ under different window scopes.

however, poses the question how to set *TH*. A simple experiment shows that a static *TH* is not good enough to keep the latency bound *LB*. We run evaluations using the traffic monitoring operator at an average inter-arrival time of cars of 200 ms, aiming to keep $LB = 1s$. With $TH = 100ms$, reactive batch scheduling more or less was able to keep *LB* when *ws* was not higher than 500 s (cf. Figure 4.4). However, at a *ws* of 600 s and 700 s, $TH = 100ms$ led to systematically wrong batch scheduling decisions; *LB* was violated by a factor of almost 100. Obviously, *TH* has to be adapted to the changing key factor values. In doing so, the feedback to change *TH* is available only after *LB* already has been violated, i.e., after a long feedback delay. The same problems apply when using the queue length peaks as a feedback parameter: The feedback delay is high. Again, using the current queue length as feedback parameter requires a suitable threshold, which in turn has to be adapted to changing key factor values.

In the face recognition operator, window scopes are much smaller. While the feedback delay of operational latency peaks is still high (46 to 410 *seconds*), the feedback delay of the queue length peaks is smaller (8 to 20 seconds; cf. Figure 4.3b). However, this does not automatically make the queue length peaks a good parameter for reactive controllers. First of all, 20 seconds is still a long time; in the real-world workloads analyzed in Section 4.4, sudden bursts demand for an even faster reaction. Second, the relation between queue length peak and operational latency peak is not trivial; the operational latency peak does not necessarily occur when the most events are in the queue, but rather when the most expensive events are in the queue. This demands for a more thorough analysis. We conclude that neither operational latency nor queue length are a reliable feedback parameter for a purely reactive batch scheduling controller.

Instead of pure feedback mechanisms, our approach uses a simple, yet powerful latency model. It takes into account feedback from operator instances, but also includes a prediction and analysis step.

4.3 Model-based Controller

The batch scheduling controller must predict whether the operational latency peak in an operator instance will be higher than *LB* when batching a new window w_{new}. To this end, we introduce a latency model. We aim to find the right balance between the complexity, the reasonable consideration of feedback from operator instances and of domain expert knowledge, and the accuracy and precision of the model.

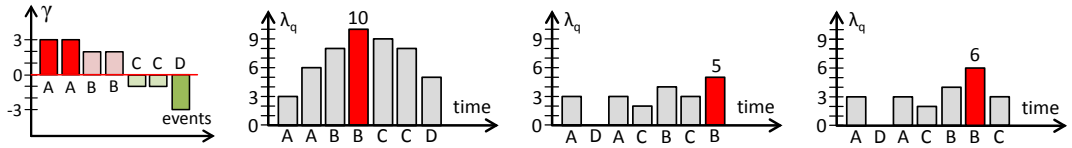
4.3.1 Basic Approach

Recall that the operational latency of an event e is built up of its queuing and processing latency: $\lambda_o(e) = \lambda_q(e) + \lambda_p(e)$. If the processing latency $\lambda_p(e)$ of an event is higher than the inter-arrival time *iat* to its successor event, this imposes additional queuing latency to the successor event. On the other hand, if $\lambda_p(e)$ is smaller than *iat*, the queuing latency of the successor event becomes smaller or even zero, i.e., e does not induce queuing latency for the successor event. In the following, we refer to the difference between λ_p and *iat* as the *gain* γ of an event: $\gamma(e) = \lambda_p(e) - iat$. If $\lambda_p(e) > iat$, we speak of a *negative gain*; else, we speak of a *positive gain*[1]. In Figure 4.5a, we provide an example. Suppose that the *iat* between events is 5 time units (TU), and the window contains 7 events: 2 events of type A impose each λ_p = 8 TU, 2 events of type B impose each λ_p = 7 TU, 2 events of type C impose each λ_p = 4 TU, and 1 event of type D imposes λ_p = 2 TU. Then, the gains of the single events are between +3 and -3 TU (+3 for type A, +2 for B, -1 for C, -3 for D).

Now, for the overall window w_{new}, the aggregated gains of the set of events with $\lambda_p(e) > iat$ are termed the *total negative gain*: $\Gamma^- = \sum \gamma(e) : e \in \mathsf{w}_{new} \wedge \lambda_p(e) > iat$. In the given example (Figure 4.5a), those are the events of type A and B; hence, $\Gamma^- = 3+3+2+2 = 10$ TU. The aggregated gains of the set of events with $\lambda_p(e) < iat$ are termed the *total positive gain*[2]: $\Gamma^+ = \sum \gamma(e) : e \in \mathsf{w}_{new} \wedge \lambda_p(e) < iat$. In the

[1] Negative gains are positive numbers and positive gains are negative numbers. The terminology refers to the impact of an event on the feasibility to schedule a window in a batch.

[2] If $\lambda_p(e) = iat$, neither negative nor positive gains occur.



(a) Latency gains. (b) W-C: Latency peak. (c) B-C: Latency peak. (d) M-C: Latency peak.

Figure 4.5: Different sequences of negative and positive gains, in worst case (WC), best case (BC) and in a medium case (MC).

given example (Figure 4.5a), those are the events of type C and D; hence, $\Gamma^+ = (-1)+(-1)+(-3) = -5$ TU.

After defining the total negative and positive gains, in the following, we analyze possible sequences of negative and positive gains and the impact on the queuing latency peak λ_q^{max}. In Figure 4.5b, first all negative gains occur, followed by all positive gains. This is the worst case with respect to λ_q^{max}; in the example sequence, λ_q^{max} = 10 TU. Note, that also any other sequence of events of types A and B would lead to the same λ_q^{max}. In the worst case, hence, $\lambda_q^{max} = \Gamma^-$. However, an interleaving between negative and positive gains is possible as well. In the examples in Figures 4.5c and 4.5d, the events with negative and positive gains interleave to a different extent. This leads to different values of λ_q^{max}, because although the queuing latency is increased by events with negative gains, events with positive gains compensate for that; a successor event of an event with positive gain faces a lower queuing latency.

The actual sequence of events with negative and positive gains in w_{new} is very difficult to predict. It would essentially correspond to predicting each single event in w_{new} and its *iat*. To account for the discussed interleaving of events with negative and positive gains, therefore, we introduce a *compensation factor* α. α allows for modeling the extent of interleaving of negative and positive gains without the need to explicitly define the sequence of events in w_{new} in the prediction: $\lambda_q^{max} = \Gamma^- + \alpha * \Gamma^+$. Taking a look at the best-case example in Figure 4.5c, we see that the negative and positive gains are maximally interleaving, hence, $\alpha = 1$. Accordingly, $\lambda_q^{max} = 10 + 1 * (-5) = 5$. Figure 4.5d exemplifies an event sequence in between the worst- and best-case: Parts of the positive gains are interleaving with the negative gains, hence, $\alpha = 0.8$. Accordingly, $\lambda_q^{max} = 10 + 0.8 * (-5) = 6$.

Please notice that the first event of w_{new} might already face a queuing latency λ_q^{init} at its arrival. This can be due to previous windows that had been scheduled to the same

operator instance. Hence, the final formula to calculate the queuing latency peak is:[3]

$$\lambda_q^{max} = \lambda_q^{init} + \Gamma^- + \alpha * \Gamma^+, \alpha \in [0,1].$$

From the queuing latency peak λ_q^{max}, the operational latency peak λ_o^{max} is calculated by adding the maximal processing latency λ_p^{max} of any event in w_{new}. This bases on the pessimistic assumption that the most expensive event occurs right at the queuing latency peak; as we do not know the event sequence, this assumption is justified by the goal to avoid underestimations of λ_o^{max}. Hence,

$$\lambda_o^{max} = \lambda_q^{max} + \lambda_p^{max}.$$

Using this latency model, the operational latency peak can be predicted, and the scheduling decision—to batch or not to batch—can be performed accordingly. In the following, we describe how the parameters of the model are predicted.

4.3.2 Prediction of Model Parameters

The proposed latency model is based on the prediction of the total sum of negative and positive gains of all events in w_{new}; i.e., it does not consider individual events, but it regards events in w_{new} as *sets* of events imposing negative or positive gains. Hence, it builds on the prediction of the set of events in w_{new}, including their processing latency λ_p and their inter-arrival time *iat*. Further, a prediction of the initial queuing latency λ_q^{init} and the compensation factor α is needed. Based on those values, the model predicts the operational latency peak. In this section, we introduce and discuss appropriate prediction methods and algorithms.

Inter-arrival time. The splitter continuously monitors the past *iat* values in a window of *mtime* time units. Our *iat* model tackles two challenges: heavy fluctuations of the *iat* around an average value (variance) and rapid changes of the average *iat* (changing trend).

Tackling the first challenge, the splitter arranges the monitored inter-arrival times in a discrete model (cf. Figure 4.6). The range of measured *iat* values is divided into a number of equally-sized *bins*. The measured *iat*s are sorted into the corresponding bin; for each bin B_i, the mean value $\overline{iat(B_i)}'$ is computed. To each bin, a $weight(B_i)$ is assigned, i.e. ratio of number of entries in the bin to total number of measurements in all bins. The number of bins controls the accuracy of the model; the minimum number of bins is 1.

[3]For the sake of readability, we did not mention in the text that $\lambda_q^{max} = \lambda_q^{init}$, if $\Gamma^- + \alpha * \Gamma^+ < 0$.

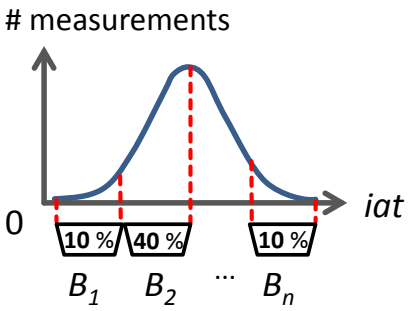


Figure 4.6: *iat* bins.

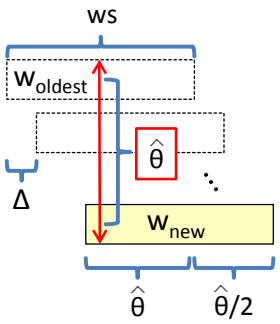


Figure 4.7: Overlap.

Tackling the second challenge, we introduce a negative bias on the monitored mean value $\overline{iat(B_i)}'$ in each bin. This way, the model accounts for changes in the average *iat* between the monitored value $\overline{iat(B_i)}'$ and the value that will occur in w_{new}. The negative bias is modeled based on a factor δ_{iat} of standard deviations σ of the monitored *iat*s, e.g., 1 standard deviation or 2 standard deviations. Then,
$iat(B_i) = \overline{iat(B_i)}' - \delta_{iat} * \sigma$.

Processing latency. In our model, λ_p depends on the overlap Θ and the processing latency in a single window λ_p^w: $\lambda_p = \Theta * \lambda_p^w$. As discussed in Section 4.2, λ_p^w depends on the event type and the position of the event in a window. Hence, first of all, our model differentiates between different event types. This design decision has two consequences: First, the prediction model of λ_p^w takes into account the type, i.e., predicts $\lambda_p^w(type)$, the in-window processing latency of events of a specific type. Second, the set of events in w_{new} is predicted with respect to the number of events of different types.

For modeling $\lambda_p^w(type)$, we propose the same methods as for modeling *iat*, using a combination of negative bias and bins. Same as in *iat* bins, in each latency bin B_l, we predict $\lambda_p^w(B_l) = \overline{\lambda_p^w(B_i)}' + \delta_{\lambda_p} * \sigma$, i.e., the measured mean in-window processing latency in the latency bin plus a factor δ_{λ_p} of standard deviations. The advantage of monitoring the current (distribution of) $\lambda_p^w(type)$ in the operator instances over building a position-dependent latency model is that we can *implicitly* incorporate the position dependency: When the (distribution of) positions of events in windows change, e.g., due to changing workload or changing window scopes, this is reflected in the monitored current (distribution of) $\lambda_p^w(type)$ values.

The overlap Θ for all events of w_{new} is modeled as the average overlap of events of w_{new} in the current batch, denoted by $\overline{\Theta}$. Predicting $\overline{\Theta}$ is performed according to the following model (cf. Figure 4.7). When w_{new} is scheduled in a batch of already opened

windows, a number of events in w_{new} has the current overlap $\hat{\Theta}$, until the oldest open window w_{oldest} in the batch closes. From closing w_{oldest} until closing w_{new}, the overlap decreases step-wise in regular intervals each time a window between w_{oldest} and w_{new} is closed. In that phase, the average overlap is $\hat{\Theta}/2$. In order to compute $\overline{\Theta}$, we weigh the ratio of events with overlap $\hat{\Theta}$ to the events with overlap $\hat{\Theta}/2$. In doing so, we assume in our model that all windows in the batch have the same window scope *ws*, and between the start of two windows there is the same shift Δ; *ws* and Δ are measured in the splitter at regular intervals to keep them up to date at each scheduling decision.

At the start of w_{new}, w_{oldest} is already open since $(\hat{\Theta}-1)*\Delta$ time units, as $\hat{\Theta}-1$ is the number of windows between w_{oldest} and w_{new} that were opened in intervals of Δ time units. Therefore, w_{oldest} stays open for $ws-(\hat{\Theta}-1)*\Delta$ more time units. When w_{oldest} closes, the phase of closing windows starts, spanning $(\hat{\Theta}-1)*\Delta$ time units. Hence, the weighed average overlap is computed as follows:

$$\overline{\Theta} = \frac{(ws-(\hat{\Theta}-1)*\Delta)*\hat{\Theta}+((\hat{\Theta}-1)*\Delta)*\hat{\Theta}/2}{ws}.$$

Number of events. For predicting the set of events in w_{new}, there are three significant factors in the model: (1) The window scope *ws*, (2) the *iat*, and (3) the ratio of different event types, denoted as $ratio(type)$, that models which percentage of events in w_{new} is of a specific *type*. These factors are gained from monitoring them in the incoming event stream in the splitter in the past *mtime* time units. To predict the total number of events in w_{new}, we again use a negative bias of δ_{iat} standard deviations $\sigma(iat)$, so that $iat = \overline{iat}' - \delta_{iat} * \sigma(iat)$. Then, the total number of events n is predicted as $n = \frac{ws}{iat}$, and the number of events of a specific *type*, denoted by $\#(type)$, is predicted as $ratio(type) * n$.

Initial queuing latency. The initial queuing latency is predicted for each operator instance separately, depending on the content of the incoming event queue. To this end, operator instances report the number of events of each type and their average overlap $\overline{\Theta}$ in the assigned windows in regular intervals to the splitter. The splitter calculates λ_q^{init} of an operator instance as the sum of the processing latencies of all reported events in its queue: $\lambda_q^{init} = \sum_{types} \#events * \overline{\Theta} * \lambda_p^{\mathsf{w}}(type)$.

Compensation factor. For modeling the compensation factor α, there are two possibilities.

First, we propose a heuristic, denoted as T-COUNT, for adapting α based on the current extent of interleaving between events with different processing latency in the incoming

stream. To this end, events are divided into two groups, a group with high processing latency, denoted by T^- and a group with low processing latency, denoted by T^+. The distinction between the groups is made based on the average $\lambda_p^{\mathsf{w}}(type)$ of the event types. The event types are sorted by their average $\lambda_p^{\mathsf{w}}(type)$. Then, the sorted list is split into two halves: The 50th quantile of event types with highest processing latency are assigned to T^-, the other event types are assigned to T^+. The splitter continuously counts in a monitoring window of temporal size *mtime*, how many events of the event types in T^-, denoted by c^-, and how many events of the event types in T^+, denoted by c^+, occur. Further, the splitter counts how often events in T^- and T^+ follow each other, i.e., the number of transitions, denoted by c^t. The maximal number of transitions is $2 * min\{c^+, c^-\}$. Trivially, the minimum number of transitions is 1. Then, α is predicted as the proportion of c^t to the maximal number of transitions: $\alpha = \frac{c^t - 1}{2 * min\{c^+, c^-\}}$.

The second alternative is that a domain expert sets a fixed or dynamic value of α based on off-line training if the characteristics of the expected workloads are known beforehand.

4.3.3 Scheduling Algorithm

Having a prediction of the events in w_{new}, including their processing latencies and inter-arrival times, the batch scheduling controller predicts the total negative and positive gains and the operational latency peak in order to schedule w_{new}. In this section, we introduce the algorithms.

Total negative and positive gains prediction. To predict Γ^- and Γ^+, the predicted processing latencies and inter-arrival times have to be combined. Each processing latency bin represents a number of events in w_{new} having a specific λ_p; each *iat* bin represents a number of events having a specific *iat*. In order to calculate the total negative and positive gain of all events, the number of events having a specific *combination* of λ_p and *iat* is predicted. To this end, events from the bin with the highest λ_p are combined with the lowest *iat*, and events with the lowest λ_p are combined with the highest *iat*. The concrete algorithm is presented in the following (cf. algorithm in Figure 4.8). First, for each type, the total number of events, $\#(type)$, is divided into latency bins according to the weights of the bins: The number of events $\#(B_l)$ in a latency bin B_l is: $\#(B_l) = \#(type) * weight(B_l)$. Then, all latency bins of all event types are sorted by their mean processing latency (highest first). The *iat* bins are sorted by the their mean *iat* (lowest first); the number of events $\#(B_i)$ in an *iat* bin B_i is computed based on

```
1: ⟨long, long⟩ predictGains ( ) begin // returns Γ⁻ and Γ⁺
2:  predict #events for each latency bin B_l :  #(B_l)
3:  sort latency bins by mean latency (highest first)
4:  predict #events for each iat bin B_i :  #(B_i)
5:  sort iat bins by mean iat (lowest first)
6:  while true do
7:      #combination ← min{#(B_l), #(B_i)}
8:      gain ← #combination * (Θ̄ * λ_p^w(B_l) − iat(B_i))
9:      if gain > 0 then
10:         Γ⁻ ← Γ⁻ + gain
11:     else
12:         Γ⁺ ← Γ⁺ + gain
13:     end if
14:     #(B_l) ← #(B_l) − #combination
15:     #(B_i) ← #(B_i) − #combination
16:     if #(B_i) = 0 then
17:         i ← i + 1 // next iat bin
18:     end if
19:     if #(B_l) = 0 then
20:         l ← l + 1 // next latency bin
21:     end if
22:     if no more bins then
23:         return ⟨Γ⁻, Γ⁺⟩
24:     end if
25:  end while
26: end function
```

Figure 4.8: Predict negative and positive gains.

```
1: OperatorInstance ω_x // current operator instance
2: void schedule ( ) begin
3:  λ_o^max ← LatencyModel.newPrediction()
4:  if λ_o^max ≤ LB then
5:      assign σ to ω_x
6:  else
7:      x ← (x + 1) MOD #op_instances // Round-Robin
8:      assign σ to ω_x
9:  end if
10: end function
```

Figure 4.9: Batch scheduling algorithm.

the total number of events, n, and the weight of the bin, $\#(B_i) = n * weight(B_i)$. Then, the numbers of events in the processing latency bins and *iat* bins are combined such that the highest processing latencies are combined with the lowest *iat*s. The algorithm iterates through the bins (lines 6 – 25): For the combination of a specific latency and *iat* bin, the gain of the events in this combination is calculated based on the processing latency and the *iat* of the bins. If the predicted gain is greater than 0, it is added to the total negative gains, else, it is added to the total positive gains. Then, the next combination of bins is processed. When the iteration went through all bins, the resulting total negative and positive gains are returned.

Operational latency peak. The operational latency peak λ_o^{max} is predicted with the formulas introduced in Section 4.3.1, taking into account the predicted parameters as described in Section 4.3.2: $\lambda_o^{max} = \lambda_q^{max} + \lambda_p^{max}$, with $\lambda_q^{max} = \lambda_q^{init} + \Gamma^- + \alpha * \Gamma^+$. In doing so, λ_p^{max} is predicted as the in-window processing latency λ_p^{w} of the most expensive event type in the most expensive latency bin, denoted $max(\lambda_p^{\mathsf{w}})$, at the average overlap: $\lambda_p^{max} = \overline{\Theta} * max(\lambda_p^{\mathsf{w}})$.

Batch Scheduling. When scheduling a new window, the controller checks whether batching it to the same operator instance the last window was assigned to would lead to a violation of *LB*. The scheduling algorithm is listed in Figure 4.9. The latency model is queried for a prediction of the operational latency peak λ_o^{max} (line 3). The predicted λ_o^{max} is compared to *LB* and a batch scheduling decision is made accordingly: If $\lambda_o^{max} \leq LB$, the window is assigned to the same instance as the last window (lines 4–5); else, it is scheduled to the next operator instance according to the Round-Robin algorithm (lines 6–8).

4.4 Evaluation

In our evaluations, we analyze the proposed batch scheduling controller in two steps. In a first step, we perform a distinct evaluation of the proposed latency model. We show the accuracy and precision of the latency model when predicting the negative gains, positive gains and latency peaks in different situations under synthetic workloads. In the second step, we measure the performance of the overall event processing system under different realistic conditions—such as inter-arrival times and latency bounds—comparing the model-based batch scheduling controller to Round-Robin and to a reactive batch scheduling algorithm. The cost of prediction is also evaluated.

Symbol	Parameter Description
iat	average inter-arrival time of events
b	batch size, i.e., number of subsequent windows scheduled to same op. instance
ws	window scope, i.e., temporal scope of a window
Γ^-, Γ^+	total negative and positive gains
α	compensation factor
λ_o, λ_q, λ_p	operational latency, queuing latency and processing latency of an event in an operator instance; $\lambda_o = \lambda_q + \lambda_p$
λ_q^{max}	queuing latency peak: $\lambda_q^{max} = \lambda_q^{init} + \Gamma^- + \alpha * \Gamma^+$
λ_q^{init}	initial queuing latency before processing the first event of a window
LB	latency bound, i.e., the peak operational latency that shall not be exceeded
RR	Round-Robin scheduling, circularly assigns one window to each operator instance
$\delta_{iat}, \delta_{\lambda_p}$	negative bias of measured *iat* or λ_p in the monitoring window, in std. deviations: e.g., $iat - \delta_{iat} * \sigma$
mtime	size of the workload monitoring window
TH	scheduling threshold of reactive baseline controller, cf. Section 4.2.2

Figure 4.10: Symbols used.

Experimental Setup and Notation. To evaluate the batch scheduling controller, we have integrated it into the existing data parallelization framework that has been introduced in Chapter 3. All experiments were performed on a computing cluster consisting of 16 homogeneous hosts with each 8 CPU cores (Intel(r) Xeon(R) CPU E5620 @ 2.40 GHz) and 24 GB memory, connected by 10-GB Ethernet links. The components of the parallelization framework were distributed among the available hosts. Symbols used in the evaluations are listed in Figure 4.10.

4.4.1 Latency Model

In the following, we evaluate the accuracy and precision of the proposed latency model. We present the evaluation in two parts: First, we evaluate the predictions of the total negative and positive gains. Based on that, we then analyze the prediction of the queuing latency peak, which depends on the prediction of negative and positive gains as well as on the compensation factor α.

Interpretation of the figures in this section. We measured both the predicted values as well as the values that actually occurred in the operator instances. In all experiment results, on the y-axis, we depict the predicted values normalized to the measured values. For example, a value of 1.0 means that the prediction exactly met the actually occurred value, a value smaller than 1.0 means that the prediction was too low (i.e., underestimation), and a value higher than 1.0 means that the prediction was too high (i.e., overestimation). All figures depict the 10th, 25th, 50th, 75th, and 90th quantiles in a "candlesticks" representation.

Negative and Positive Gains

In analyzing the prediction of Γ^- and Γ^+, we run evaluations on synthetic workloads. Using synthetic workloads allows us to perform measurements in controlled situations where all of the parameters are well-known and completely under our control. This is not the case in real-world workloads, as used in the evaluation of the overall event processing system in Section 4.4.2. For the face recognition operator, we created a synthetic stream of face events (i.e. images containing a person's face). Each 2 seconds, a burst of 4 face events with an inter-arrival time of 10 ms was created, which resembles 4 persons in front of a camera that captures a picture each 2 seconds. The query events were generated with a fixed rate of 1 query per second, so that each second, one

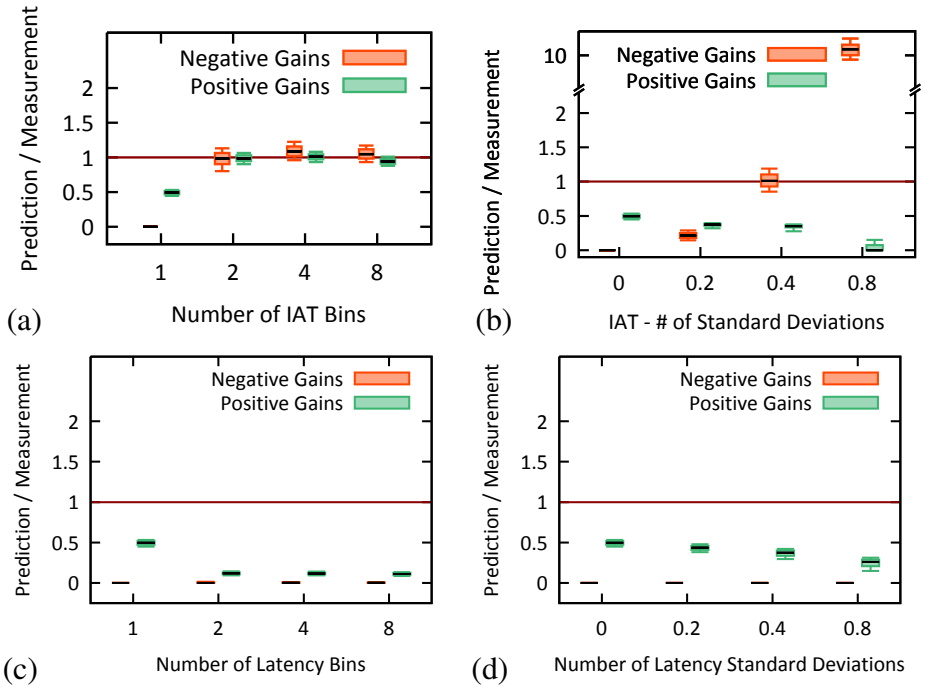


Figure 4.11: Face recognition operator at $ws = 10s$: prediction of negative and positive gains. $b = 1$.

new window was started. For the traffic monitoring operator, we created a workload trace with an average inter-arrival time of events of 100 ms that follows an exponential distribution, which resembles 5 cars per second passing each road checkpoint.

Figure 4.11a shows evaluations of the prediction of negative and positive gains in the face recognition operator for a single window (b = 1) using a different number of *iat* bins. If only 1 bin is used, the predictions of Γ^- and Γ^+ are poor. With a growing number of *iat* bins, the latency model becomes more accurate: With 2, 4 or 8 bins, the predictions of both Γ^- and Γ^+ are very accurate. 2 bins are sufficient, as the workload is also divided into two phases: face events arrive in bursts, and in between the bursts, no face events arrive. In contrast to the effect of *iat* bins, using a negative bias of δ_{iat} standard deviations does not make the predictions more accurate, but more *pessimistic* (cf. 4.11b): The higher δ_{iat} is, the higher is the predicted Γ^-, but the lower is the predicted Γ^+. Further, we evaluate the impact of using bins and pessimistic bias for processing latency. As shown in Figures 4.11c and 4.11d, neither of the two strategies had positive impact on the accuracy of the latency model. This is because the processing latency of single events in single windows does not fluctuate very much in the face recognition operator.

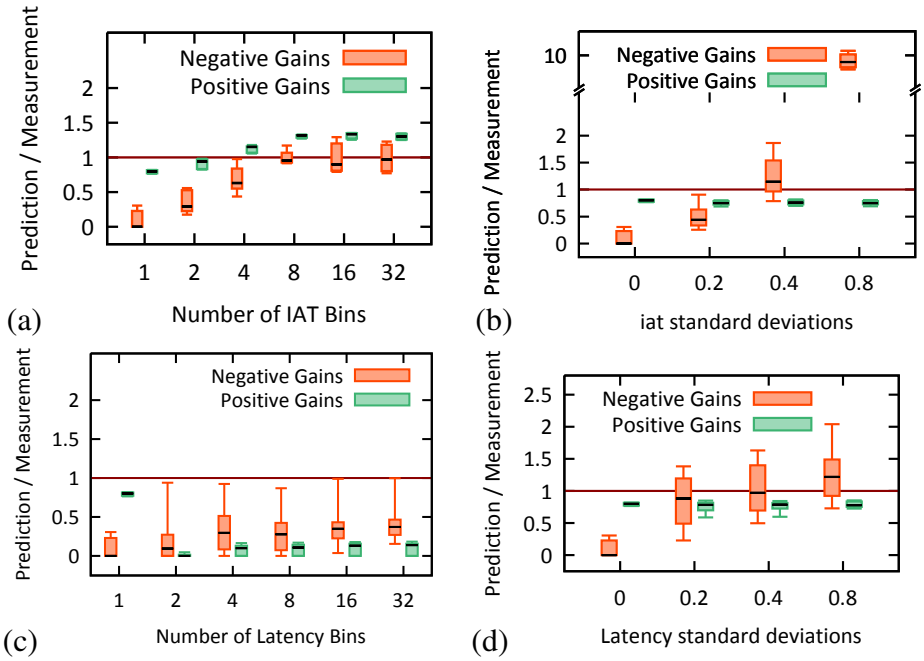


Figure 4.12: Traffic monitoring operator at $ws = 900s$: prediction of negative and positive gains. $b = 1000$.

We evaluated the latency model also with the traffic monitoring operator when batching 1,000 windows (b = 1,000). Same as in the face recognition operator, employing *iat* bins quickly improves the prediction accuracy (cf. Figure 4.12a). Further, using a negative bias of δ_{iat} standard deviations makes the prediction more pessimistic (cf. 4.12b). Concerning processing latency, employing latency bins improves the accuracy of the latency model only slightly (cf. Figure 4.12c). Even though the processing latency in the traffic monitoring operator is position-dependent, the occurrence of negative and positive gains is still dominated by the *iat*; hence, the usage of latency bins alone does not lead to satisfactory results. Employing a negative bias of δ_{λ_p} standard deviations on processing latency again makes the latency model more pessimistic (cf. Figure 4.12d).

We also tested both scenarios with a higher batch size. In the face recognition operator at $b = 4$, employing *iat* bins leads to the same improvements of the model accuracy (cf. Figure 4.13a). In the traffic monitoring operator at $b = 2000$, even at only one *iat* bin, the accuracy is already high and adding more bins does not improve the model. This is because at a high batch size, the overlap of windows is high, and hence, the processing latency of events—especially those of type *L2* (cf. Section 4.2.1)—is high as well. Basically, that means that most of the events of type *L2* will generate a negative gain,

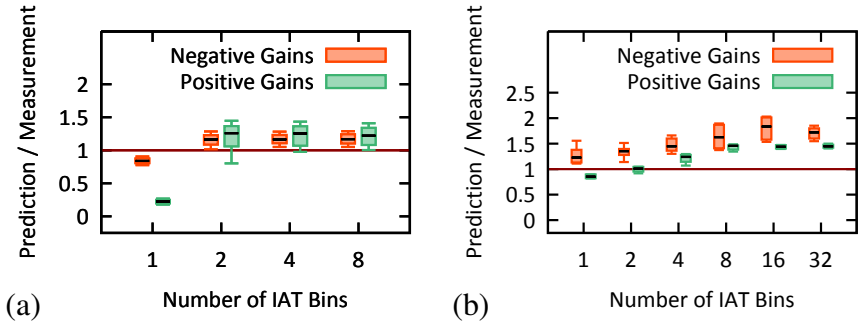


Figure 4.13: Higher batch sizes. (a) Face recognition, $b = 4$. (b) Traffic monitoring, $b = 2000$.

no matter if the specific *iat* of an event is high or low. In other words, the fluctuations of *iat* do not dominate the total negative and positive gains any longer; therefore, more *iat* bins do not improve the model.

Queuing Latency Peak

Recall that the queuing latency peak is predicted based on the total negative and positive gains and the compensation factor α: $\lambda_q^{max} = \lambda_q^{init} + \Gamma^- + \alpha * \Gamma^+$. We show on the examples of the face recognition operator and the traffic monitoring operator that our proposed T-COUNT heuristic provides a suitable, slightly pessimistic estimation of α such that no under-estimation of queuing latency peak occurs. Additionally, we evaluate the prediction of the initial queuing latency λ_q^{init}. Following our observations from Section 4.4.1, we employ the latency model with 2 *iat* bins, so that the predictions of Γ^- and Γ^+ are accurate.

For the face recognition operator, we see in Figure 4.14 that the T-COUNT heuristic leads to a good overall estimation of λ_q^{max}. In predicting λ_q^{init}, fluctuations are caused by events in the network that have not yet arrived in the queue of an operator instance and are not considered in the feedback to the splitter. However, the impact of this issue on the prediction of λ_q^{max} is small, as λ_q^{max} is dominated by the negative and positive gains.

In the traffic monitoring operator, at $b = 2000$ and $ws = 900s$, T-COUNT also guarantees that λ_q^{max} is not underestimated (cf. Figure 4.15). It is worth to mention that λ_q^{init} was at 0 all the time, that means, even when scheduling the 2000th window in the batch, the incoming queue of an operator instance was still empty; hence, we omit the (trivial) prediction of λ_q^{init} in the figure. Queuing only happens when the events with

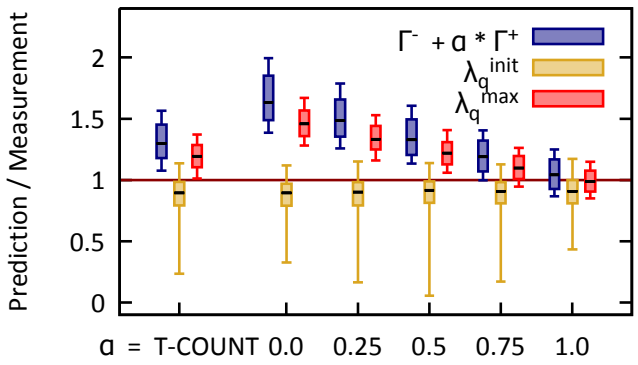


Figure 4.14: Predictions of queuing latency peak. Face recognition operator, $b = 4$, $ws = 10s$.

high positions in the windows of the batch are processed. This effect supports our argumentation from Section 4.2.2 that pure feedback-based scheduling is not applicable to the batch scheduling problem.

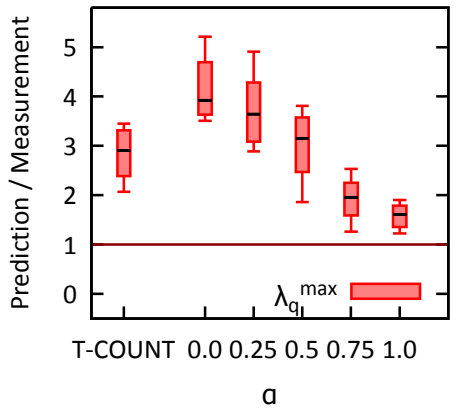


Figure 4.15: Predictions of queuing latency peak. Traffic monitoring operator, $b = 2000$, $ws = 900s$.

Besides the T-COUNT heuristic, we also systematically evaluated the impact of fixed values of α on the prediction of λ_q^{max}. As can be seen in Figures 4.14 and 4.15, using different fixed values leads to different degrees of over- or underestimations of λ_q^{max}. Off-line profiling can be used in order to develop pessimistic or optimistic models to set α, when the characteristics of the workload are well-known before system deployment.

4.4.2 Overall Event Processing System

We compare our model-based batch scheduling controller to two baseline scheduling algorithms: Round-Robin scheduling and latency-reactive scheduling. Round-Robin aims for good load balancing but disregards communication overhead; it is the standard scheduling algorithm used in window-based data parallelization systems such as [MKR15]. Latency-reactive scheduling, as described in Section 4.2.2, batches windows to an operator instance until its operational latency exceeds a threshold *TH*. It is used as a latency-aware baseline algorithm to compete against our model-based controller.

Traffic Monitoring Scenario. In our dynamic traffic monitoring scenario, we modeled the inter-arrival time of vehicles as an exponential distribution with an average value following a sinusoidal curve between 2000 ms and 200 ms. Following the evaluation of the latency model in Section 4.4.1, we set-up the controller to use 8 *iat* bins and a tumbling monitoring window with $mtime = 60s$. To account for the position-dependency of the operator and the rapdily changing workload, we add a pessimistic bias of $\delta_{\lambda_p} = 2$ standard deviations on the monitored processing latency and $\delta_{iat} = 0.75$ standard deviations on the monitored *iat*. In all experiments, the parallelization degree, i.e., number of operator instances, was fixed at 8. Each experiment was running for 5 hours.

At a window scope of 500 seconds, Round-Robin scheduling resulted in an operational latency peak of 200 ms (cf. Figure 4.16a). 724,464 events have been transmitted between the splitter and the operator instances (cf. Figure 4.16b). We ran the same experiment using our batch scheduling controller allowing for 2.5, 5 and 10 times higher operational latency peaks than yielded in Round-Robin: 500 ms, 1 s and 2 s. As shown in Figure 4.16a, *LB* was kept. ***The communication overhead was reduced by 53 %, 59 % and 64 %, respectively*** (cf. Figure 4.16b). We compared this performance to the latency-reactive scheduler described in Section 4.2.2; the reactive scheduler batches windows to an operator instance until it reports a current operational latency of more than $TH = 100ms$. The operational latency and communication overhead was very similar to model-based scheduling at $LB = 2s$; however, the tail of the latency distribution is much longer, leading to 50 % higher operational latency peaks. This indicates that the reactive scheduler erratically batches too many windows, leading to a less predictable latency of the operator instances than when the model-based controller is used.

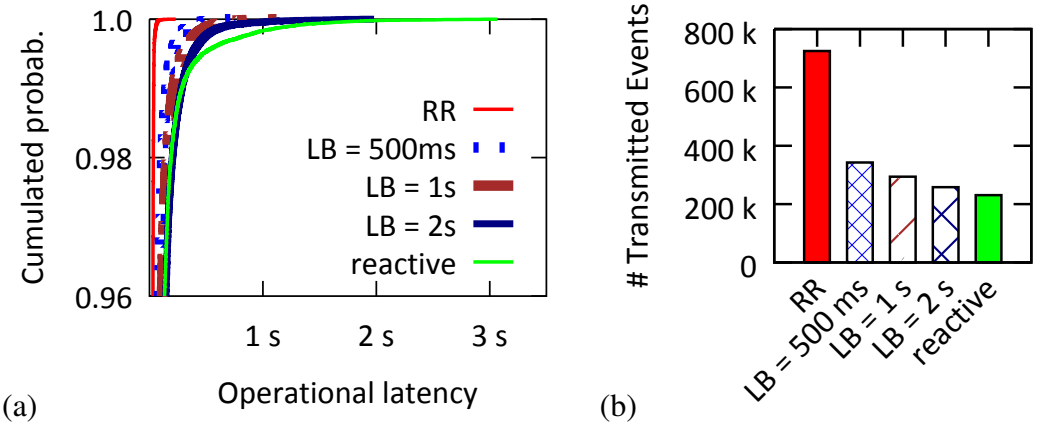


Figure 4.16: Traffic monitoring operator. (a) Operational latency. (b) Communication cost.

Face Recognition Scenario. With the dynamic face recognition scenario, we evaluate the system behavior at a highly bursty *real-world workload.* A *real video stream* from a camera installed on campus—capturing 1 frame each 2 seconds—is processed by a face detection operator and the detected faces are streamed to the face recognition operator. Simulating users of a face recognition application, the arrival of new queries is modeled as an exponential distribution with an average inter-arrival time of 2 seconds. The face recognition operator detects whether the queried person is in the face event stream, using a window scope of $ws = 10s$. Each experiment ran for 150 minutes. According to the insights we gained from the evaluation of the latency model in Section 4.4.1, we set-up the controller to use 2 *iat* bins. Further, we set $mtime = 10s$ (tumbling window) and $\delta_{iat} = 1.0$ standard deviations to account for the changing *iat*.

For Round-Robin scheduling, we measured an operational latency peak of 6 seconds (cf. Figure 4.17a). 68,412 events have been transmitted between the splitter and the operator instances (cf. Figure 4.17b). We ran the same experiment using our batch scheduling controller allowing for 2.5, 5 and 10 times higher operational latency peaks than yielded in Round-Robin: 15 s, 30 s and 60 s. The latency bounds are kept in all tested settings (cf. Figure 4.17a). ***The communication overhead was reduced by 14 %, 31 % and 76 %, respectively*** (cf. Figure 4.17b). We compared this performance to the latency-reactive scheduler described in Section 4.2.2 with $TH = 6s$. The operational latency peaks were 15 % higher than with the model-based controller at $LB = 60s$, while the communication overhead was 14 % higher as well. With a higher threshold *TH*, the reactive scheduler would induce even higher latency peaks, while with a lower *TH*, it would induce an even higher communication overhead; hence, the model-based

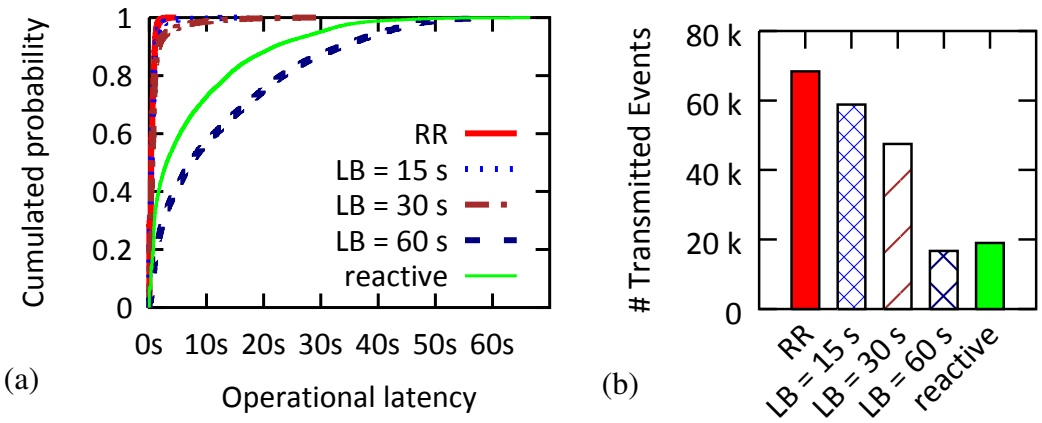


Figure 4.17: Face recognition operator. (a) Operational latency. (b) Communication cost.

controller is more effective, no matter how the reactive scheduler's threshold *TH* is set up.

In summary, model-based batch scheduling is effective in trading communication overhead against operational latency. In comparison, reactive scheduling is less predictable and effective than model-based scheduling; it might still be useful in cases where a simple best-effort batching approach is sufficient, but should not be used when latency bounds must be enforced.

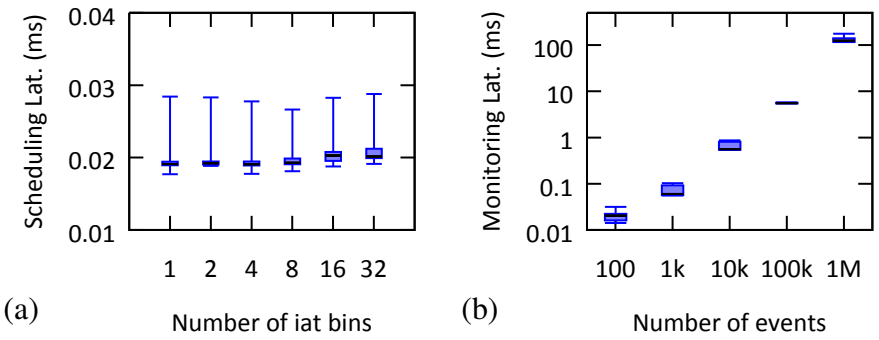


Figure 4.18: Latency of (a) scheduling and (b) updating statistics.

Scalability. We evaluate the scalability of our approach in two aspects. First, the *scheduling latency*, i.e., the time between the detection of the start of a new window and the scheduling decision (cf. Algorithm in Figure 4.9). It includes predicting the negative and positive gains (cf. Algorithm in Figure 4.8), whose complexity is determined by the granularity of the latency model, i.e., the number of bins used in the

model. We measured a very low scheduling latency of in average 0.02 ms for up to 32 bins used (cf. Figure 4.18a), which is the maximal number of bins needed in any of the scenarios that we have tested (cf. Section 4.4.1). For comparison, the median scheduling latency in reactive scheduling and in Round-Robin scheduling was both 0.004 ms. There is a small overhead for the model-based batch scheduling controller involved compared to the simple strategies. However, this is not significant in most scenarios; if scheduling would be a throughput bottleneck in the splitter, the frequency of predicting negative and positive gains could be adapted (i.e., not predicting fresh gains at each single scheduling decision) to mitigate such situations.

Second, we evaluate the time needed to update the latency model with new statistics from the monitoring window, i.e., the *monitoring latency*. This comprises recomputing the weights, average values and standard deviations of the bins. Using 32 bins, we measured a linear growth with the number of events in the monitoring window (cf. Figure 4.18b). At 1,000,000 events in a monitoring window, updating the statistics took between 100 and 200 ms, which is a reasonable time to adapt the model to changes in the workload.

4.5 Related Work

In related work, there have been addressed different problems of assigning batches of individual events to instances of stream processing operators. Das et al. [DZSS14] propose a reactive controller in order to batch a minimal number of events to an operator such that the throughput is sufficiently high to process the current workload. In their processing model, operators can aggregate larger sets of events more efficiently, so that the throughput of operators grows with the batch size. A similar problem had been studied before by Carney et al. [CcR+03]. Micro-batching, as used, e.g., in Spark Streaming [ZDL+12], provides efficient failure recovery and batch-like programming paradigms by handling streaming events as a series of fixed-sized batches. Unlike in this paper, in all of these approaches, batches are composed of individual events and not of overlapping windows. Balkesen and Tatbul [BT11] recognize the trade-off of communication overhead to latency in operator instances when scheduling overlapping windows. Their analytical cost model assumes fixed processing latency of an event in a window and fixed count-based or time-based window size and slide. Further, it does not consider inter-arrival times. Hence, it is not suitable for solving the batch scheduling problem in data-parallel CEP operators.

Elasticity in data-parallel stream processing, i.e., adapting the number of operator instances to changing workloads, is a complementary problem. Existing solutions that apply latency models often base on the assumption of fair load balancing [DMM16, MKR15, LJK15]; batch scheduling defeats this assumption, deliberately inducing a controlled load imbalance. How to use the proposed latency model of the batch scheduling controller for elasticity control is an interesting research question for future work.

Other latency models for CEP operators have been proposed. The Mace metrics from Chandramouli et al. [CGB+11] for latency estimation in a CEP middleware proposes an analytical model. However, it assumes the usage of a specific scheduling algorithm—which is not a batch scheduling algorithm. In the latency model of Zeitler and Risch, a fixed processing latency of each event is assumed [ZR11]; our latency model differentiates between different event types and takes into account the overlap of windows.

Batching is also applied in other fields, like graph processing [XWB+13] and column data-stores [LFV+12], where it is often preferable to process or store data in batches instead of handling each single tuple separately. However, typically, optimal batch sizes are predefined, e.g., by cache sizes, so that fixed batch sizes are employed.

Scheduling algorithms in non-parallel CEP optimize the utilization of resources like CPU and memory [KLB08, BBD+04] without taking into account batching of overlapping data sets.

4.6 Conclusion

In this chapter, we have tackled the problem to batch as many subsequent overlapping windows as possible to the same operator instance in data-parallel CEP operators subject to the constraint that the operational latency in the operator instance must not exceed a given latency bound. As the batch scheduling decisions are made on open windows, a long feedback delay between the decisions and their impact on feedback parameters is induced, making reactive scheduling approaches infeasible. Instead, we have proposed a model-based controller. Evaluations show that the controller batches an optimal amount of windows even at bursty workloads. This way, the bandwidth consumption of data-parallel CEP operators can be reduced significantly.

5

Supporting Consumption Policies

In the previous Chapters 3 and 4, we have seen that window-based, data-parallel CEP systems split the incoming event streams into independently processable windows that capture the temporal relations between single events imposed by the queried event pattern. The windows are processed in parallel by a number of identical operator instances. An event can be part of different windows, so that windows may overlap.

A crucial question in overlapping windows is whether an event can be used in multiple pattern instances or not. In many cases, it is preferable to *consume* an event once it is part of a pattern instance. In particular, this means to not use the same event for the detection of further pattern instances in other windows. This way, semantic ambiguities and inconsistencies in the complex events that are emitted can be resolved or prevented. The problem tackled in this paper is that event consumptions impose dependencies between the different windows and thus, prevent their parallel processing. When the same event is processed in parallel in two different windows, consuming it in the first window also consumes it from the second window; hence, there is a dependency between both windows, which can hinder their parallel processing. Understanding that problem, it is no surprise that existing parallel implementations of CEP systems [CM12b, BDWT13, MKR15] do not support event consumptions, whereas sequential systems often do [CM94, AE04, CM12a]. This limits the scalability of operators that impose event consumptions. Moreover, it even impedes event consumptions from their further development in academia and industry, as in times of Big Data and Internet of Things, parallel CEP systems are becoming the gold standard.

Toward this end, in this chapter, material published in [MST+17] is presented. In particular, we propose a speculative processing method that allows for parallel processing

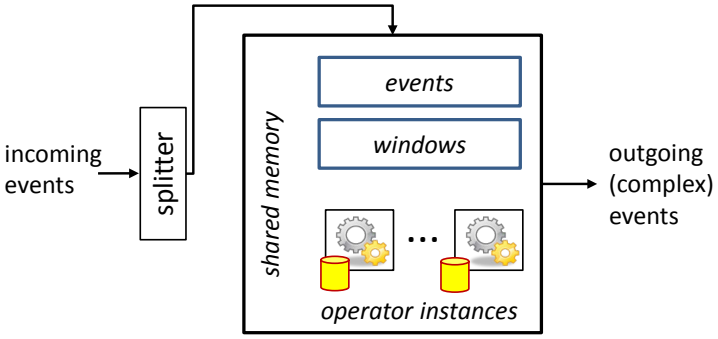


Figure 5.1: Data parallelization framework.

of window-based CEP operators in case of event consumptions. The basic idea is to speculate in each window which events are consumed in the previous windows—instead of waiting until the previous windows are completely processed. This way, multiple overlapping windows can be processed in parallel despite inter-window dependencies. To this end, we propose the SPECTRE (SPECulaTive Runtime Environment) framework, comprising the following contributions: (1) A speculative processing concept that allows the execution of multiple versions of multiple windows using different event sets in parallel. (2) A probabilistic model to process always those window versions that have the highest probability to be correct. (3) Extensive evaluations that show the scalability with a growing number of CPU cores.

5.1 System Model and Problem Analysis

Here, we first discuss extensions of the system model that has been introduced in Chapter 2. Then, we analyze the challenges for parallel event processing that are posed by event consumptions.

5.1.1 Extensions of the System Model

We assume a shared memory (multi-core) architecture, where the splitter and operator instances are executed by independent threads running on dedicated CPU cores (cf. Figure 5.1). We assume that the underlying system can provide $k+1$ threads, so that 1 thread is pinned to the splitter and k threads are pinned to the operator instances. In the rest of this paper, we do not differentiate between operator instances (i.e., instances of

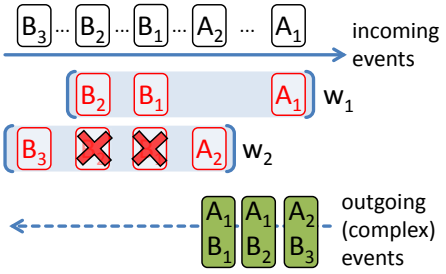


Figure 5.2: Q_E with selection policy "earliest A, each B" and consumption policy "selected B".

the pattern detection logic) and the threads that execute them—we simply refer to both as operator instances.

As mentioned above, we follow a *window-based* data parallelization approach. The incoming event streams are partitioned into windows that capture (temporal) relations defined in the queried pattern. The windows are assigned with increasing window IDs and their boundaries are stored in the shared memory (e.g., "w_i from event X to event Y").

The splitter periodically schedules to each operator instance a specific window for processing. The operator instances can hold local state of the processing in shared memory, e.g., partial pattern matches detected in the assigned window. This allows a specific window to be processed by any operator instance at any time; in particular, the processing of a window can be interrupted for some time and resumed later by the same or a different operator instance.

In this chapter, we assume that CEP operators implement consumption policies. Notice that when a complex event is detected, all constituent events of the event pattern are checked against the consumption policy. Then, all events defined by the consumption policy are consumed as a whole. This implies that events are not consumed while they only build a *partial match*, but only when the match is completed and a complex event is produced. This inherent property is independent of the concrete selection and consumption policy.

5.1.2 Challenges and Goal

In systems without consumptions, processing of a window cannot impact the events within another window, i.e., in principle each pair of windows can be processed in

parallel. However, event consumptions impose a dependency between the windows, restricting parallelism, as we discuss in the following.

Recall the example query Q_E from Section 2.1. The queried search pattern was a sequence of events of type A and B, where the event of type B follows the event of type A within a time window of 1 minute. In Section 2.1, we had discussed that depending on the selection and consumption policy of the query, different complex events will be produced. Here, we want to make clear the problems that event consumption poses on parallel processing of overlapping windows.

In Figure 5.2, we provide an example where the selection policy is "earliest A, each B" and the consumption policy is "selected B". In the first window w_1, A_1 and B_1 build a complex event $\frac{A_1}{B_1}$, such that B_1 is consumed; furthermore, A_1 and B_2 build a complex event $\frac{A_1}{B_2}$, such that B_2 is consumed. If w_1 and w_2 are processed in parallel, the consumption of B_1 and B_2 in w_1 might not be known in w_2, so that B_1 and B_2 are erroneously processed in w_2, too, leading to inconsistent results. To prevent anomalies due to concurrent processing, w_2 can only be processed after the consumptions in w_1 are known. When the event patterns are more complex than in the given minimal working example of Q_E, the dependencies become hard to control. For instance, if the pattern requires 3 rising stock quotes of B in a sequence, the completion of the pattern in w_1—and hence, the event consumptions—might be unsure until w_1 is completely processed. If 2 events of type B with rising quotes have already been detected in w_1, the completion of the pattern depends on whether a third B occurs; this might only be known at the end of w_1. The standard procedure to deal with data dependencies is to wait with processing w_2 until w_1 is completely processed and hence, all consumptions in w_1 are known. This, however, impedes the parallel processing of overlapping windows.

In this chapter, we aim to develop a framework to enable parallel processing of all CEP operators, regardless of their selection and consumption policy. To this end, we develop a speculative processing method that overcomes the data dependencies imposed by event consumptions, so that data-parallel processing becomes possible. The framework shall deliver exactly those complex events that would be produced in sequential processing; in particular, no false-positives and false-negatives shall occur.

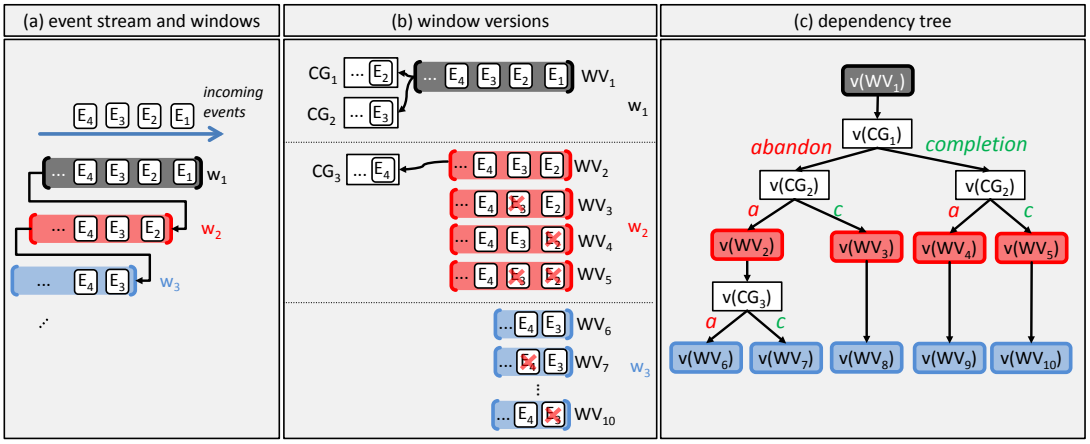


Figure 5.3: Consumption Problem: (a) Structural View. (b) Processing View. (c) Management View.

5.2 The SPECTRE System

To tackle the dependencies between different windows imposed by event consumptions, we propose the SPECTRE (SPECulaTive Runtime Environment) system, a highly parallel framework for CEP operators. SPECTRE aims to detect the dependencies between different windows and to resolve them by means of speculative execution.

This section is organized as follows. In Section 5.2.1, we introduce the speculative processing approach we follow in SPECTRE. It is based on creating multiple speculative window versions in order to resolve inter-dependencies between windows. Based on that concept, in Section 5.2.2, we explain how SPECTRE determines and schedules the k "best" window versions to k operator instances for parallel processing. Finally, in Section 5.2.3, we provide details on how the k operator instances perform the parallel processing of the assigned window versions.

5.2.1 Speculation Approach

As pointed out above, operators process their incoming data stream based on windows. In particular, operators search for queried patterns to occur in the sequence of events comprised by a window. Windows can overlap, i.e., a pair of windows might have a sequence of events in common. The windows of an operator are totally ordered according to their start events. We call a window, say w_j, a successor of another window,

w_i, iff the start event of w_i occurs before the start event of w_j in the corresponding event stream. For example, in Figure 5.3(a), w_1 starts earlier than w_2; hence, w_2 is a successor of w_1. In the same way, w_3 is a successor both of w_2 and of w_1.

Now, we can define a *consumption dependency* (or *dependency* for short) between windows. Roughly speaking, a window w_j depends on another window w_i, if the consumption of some events in w_i might affect the processing of window w_j. Formally, we define that w_j depends on w_i iff w_j is a successor of w_i and w_j overlaps with w_i. For example, in Figure 5.3(a), w_2 depends on w_1, and w_3 depends both on w_2 and on w_1.

Now, we will introduce the concept of a *consumption group*. A consumption group is maintained for each partial match of a search pattern found in a window. It records all events of this window that need to be consumed if the partial match becomes a total match, i.e., the corresponding search pattern is eventually detected in the window. Let's assume that an operator is acting on some window w. Whenever the operator processes an event starting a new partial match of some search pattern, it *creates* a new consumption group associated with w. When it processes an event that completes a pattern, it *completes* the corresponding consumption group. On the other hand, a consumption group is *abandoned* if the corresponding pattern cannot be completed anymore. Consequently, while processing the events of a window, multiple consumption groups can be created that are associated with w. However, all of them will be completed or abandoned at the latest when processing of w is finished.

While acting upon w, the operator adds events to be potentially consumed to the consumption groups associated with w, in conformance with the specified consumption policy. When a consumption group is completed, all events contained in this group are consumed together. If the consumption group is abandoned instead, it is just dropped and no events are consumed.

For example, let us assume that a query for pattern of a sequence of three events of type A, B and C in a window of time scope 1 minute, is processed by an operator. Let us further assume the consumption policy is set to consume all participating events in case of a pattern match. When detecting an event of type A, say A_1, in a window, the operator creates a new consumption group. The first event of type B, B_1, is added to the consumption group. If the window ends (i.e., 1 minute has passed) and no event of type C is detected, the consumption group is abandoned and no events are consumed in the window. If an event of type C, say C_1, occurs after B_1 and within the window scope, the consumption group is completed, and all three events participating in the pattern match, A_1, B_1 and C_1, are consumed together.

At the time a consumption group is created that is associated with window w, it is unknown whether the corresponding pattern will eventually be completed in w. Clearly, the outcome of the consumption group (complete or abandon) might affect events of all windows that depend on w. One way to handle this uncertainty is to defer the processing of all depending windows until the consumption group terminates (completed or abandoned). However, in general this amounts to processing all windows sequentially. The approach that we follow in SPECTRE is to generate two window versions for each window depending on w, one version assuming that the consumption group will be completed and the other one assuming the consumption group will be abandoned. These window versions can then be processed in parallel to w. Once the outcome of the consumption group is known, i.e., completed or abandoned, processing continues on the corresponding window versions that assume the correct outcome while the other window versions that assume the wrong outcome are just dropped. Obviously, this approach allows for processing dependent windows in parallel even in the presence of event consumptions.

With this approach, windows that depend on other windows may have multiple versions that depend on the outcome of the associated consumption groups. In principle there is a window version for any combination of the complete and abandon case of the consumption groups that a window depends upon. When one of these consumption group is abandoned, all window versions assuming this consumption group to complete can be dropped, and vice versa.

To capture the dependency between consumption groups and window versions, we introduce the concept of a *dependency tree*. There is an individual dependency tree for each independent window, i.e., each window that does not depend on any other window according to our definition above. The vertices of the dependency tree are window versions or consumption groups, while the directed edges of the tree specify the dependencies between them. The root of the dependency tree is the only version of an independent window—by definition, there is only one such version.

The vertex of a window version WV, say v(WV), has at most one child. The sub-hierarchy rooted by this child includes all versions of windows depending on WV, if any. We will denote this sub-hierarchy as v(WV)'s subtree. The subtree is rooted by a consumption group if a consumption group is associated with v(WV). Otherwise the root of the subtree is a window version directly dependent on v(WV), if any.

A vertex representing a consumption group CG, say v(CG), always has two children, one for each possible outcome of CG (completed or abandoned). The so-called com-

pletion edge of v(*CG*) links the subtree of window versions for which completion of *CG* is assumed, whereas the so-called abandon edge of v(*CG*) links the subtree of window versions which assume *CG* to be abandoned. That is, all window versions that can be reached via v(*CG*)'s completion edge do not include any event included in *CG*, while events in *CG* have no effect on window versions linked by v(*CG*)'s abandon edge.

When a consumption group *CG* associated with a window version *WV* is created, the following is performed: v(*CG*) is added as a new child of v(*WV*) to the dependency tree. The old subtree of v(*WV*) is linked by v(*CG*)'s abandon edge, while a modified copy of the subtree is linked by v(*CG*)'s completion edge. The modification makes sure that no events included in *CG* occur in the window versions of the subtree linked by v(*CG*)'s completion edge. In other words, for each window version existing in v(*WV*)'s old dependent versions subtree, a copy that *suppresses* all events listed in *CG* is added. Therefore, each new consumption group associated with v(*WV*) doubles the window versions in v(*WV*)'s subtree.

Algorithms for dependency tree management: In the following, we formalize the algorithms for the management of the dependency tree. There are three different cases that require a modification of the dependency tree: (1) a new dependent window is opened, (2) a new consumption group associated to a window version is created, (3) an existing consumption group is completed or abandoned.

New dependent window. When a new window w_{new} is opened that depends on another window w_x, for every leaf vertex of the dependency tree rooted by the window version of w_x, new window versions are created as child vertices. The corresponding algorithm is listed in Figure 5.4, lines 1–10.

Example: In Figure 5.3, at the start of w_3, new window versions (WV_6 to WV_{10}) of w_3 are created and the corresponding vertices (v(WV_6) to v(WV_{10})) are attached to all leaf nodes of the dependency tree rooted by the window version of w_1. If a leaf vertex is a consumption group *CG*, two window versions of w_3 are created and attached (a version for completion of *CG*, and a version for abandoning of *CG*); if a leaf vertex is a window version, one window version of w_3 is created and attached.

Consumption group created. Recall that when a consumption group *CG* associated with a window version *WV* is created, the old subtree of v(*WV*) is linked by v(*CG*)'s abandon edge, while a modified copy of the subtree is linked by v(*CG*)'s completion edge. The corresponding algorithm is listed in Figure 5.4, lines 12–16.

```
1:  newWindow ( ) begin
2:   for each leafVertex ∈ dependencytree do
3:       if leafVertex is window version then
4:           leafVertex.child ← new v(WV)
5:       else // else, it is a Consumption Group
6:           leafVertex.completionEdge ← new v(WV)
7:           leafVertex.abandonEdge ← new v(WV)
8:       end if
9:   end for
10: end function
11:
12:  consumptionGroupCreated (CGroup CG, WinVersion WV) begin
13:   create a modified copy of the subtree attached to v(WV)
14:   v(CG).completionEdge ←v(WV).modifiedSubtree
15:   v(CG).abandonEdge ←v(WV).originalSubtree
16: end function
17:
18:  consumptionGroupCompleted (CGroup CG) begin
19:   v(CG).abandonEdge ← null
20:   v(CG).parent.child ←v(CG).completionEdge
21: end function
22:
23:  consumptionGroupAbandoned (CGroup CG) begin
24:   v(CG).completionEdge ← null
25:   v(CG).parent.child ←v(CG).abandonEdge
26: end function
```

Figure 5.4: Algorithms for managing the dependency tree.

Example: In Figure 5.3, WV_2 creates CG_3. Then, v(CG_3) is attached as a new child to v(WV_2), and the former child, v(WV_6), becomes the root of the unmodified subtree of v(CG_3). For all window versions in the unmodified subtree of v(CG_3), a new alternative version is created that assumes that CG_3 will be completed. Suppose CG_3 contains event E_4. Then, window version WV_6 (from the unmodified subtree) contains event E_4, whereas the alternative window version WV_7 (from the modified subtree) suppresses event E_4.

Consumption group completed / abandoned. When a consumption group is completed or abandoned, the respective opposite abandon or completion path of that consumption group is removed from the dependency tree. There are two different reasons why a consumption group is abandoned: (1) Due to the termination of the corresponding window version/end of window, or (2) due to a condition from a negation statement being fulfilled. For instance, a pattern specification of a sequence of events of type *A* and *B* can define that no event of type *C* shall occur between the *A* and *B* events. If a consumption group is opened with an *A* event, the occurrence of a *C* event would trigger the consumption group to be abandoned as the pattern instance cannot be completed any more, even if a *B* event would occur later. The algorithms for subtree removal are listed in Figure 5.4, lines 18–26.

To be able to process k window versions in parallel we obviously need k operator instances. That means, that typically only a small fraction of all possible window versions can be considered for speculative processing. To be able select the k most promising window versions, we need a method for predicting the probability of possible window versions to survive (i.e., not to be dropped). In Section 5.2.2, we propose a scheme for scheduling the k most promising window versions on a collection of k operator instances.

5.2.2 Selecting and Scheduling the Top-k Window Versions

The intuition behind SPECTRE is to predict the k "best" speculative window versions and schedule them for parallel processing on k operator instances. To determine the top-k window versions, SPECTRE periodically determines the k window versions with the highest probability to survive in the entire dependency tree. In other words, SPECTRE does not create and schedule windows, as assumed in Section 5.1.1, but window versions; in doing so, multiple versions of the same window can be scheduled to different operator instances in parallel.

Whether or not a window version *WV* survives depends on the outcome of the preceding consumption groups, i.e. the consumption groups on the path from *WV* to the root of the dependency tree. In the following, we will denote this path as *WV*'s *root path*. Remember, each vertex representing a consumption group has two outgoing edges, a complete and an abandon edge. We say that the complete or abandon edge of a consumption group, say *CG*, becomes *valid* when *CG* is completed or abandoned, respectively. Once one of these edges becomes valid, the other one turns *invalid*. Consequently, *WV* survives only if all abandon and complete edges on its root path eventually become valid, i.e., *WV* is dropped if at least one of these edges turns invalid.

The probability of *WV* to survive depends on the completion probabilities of the consumption groups on *WV*'s root path. The survival probability of *WV*, denoted as $SP(WV)$ is determined as follows: Let $P(CG)$ be the probability that *CG* is completed. Moreover, let CG_c and CG_a be the set of consumption groups that contribute a complete and abandon edge to *WV*'s root path, respectively. Then[1], $SP(WV) = \prod_{c \in CG_c} P(c) \times \prod_{c' \in CG_a} (1 - P(c'))$.

Prediction Model

Now, we discuss how we predict the completion probability of a consumption group. Generally, we observe that the probability that a consumption group is completed equals to the probability that the underlying partial match for a search pattern is completed. Our scheme for predicting the completion probability $P(CG)$ of a consumption group *CG* at a given time takes into account two factors: (1) The inverse degree of completion, i.e., how many more events are at least required in order to complete the pattern—denoted by δ—and (2) the expected number of events left in the window, denoted by n. If δ is low and many events are still expected to occur in the window, the probability of completion is high. On the other hand, if δ is high and only very few events are still expected in the window, the probability of completion is low. In the following, we describe how the probabilistic model is built and updated at system run-time.

[1]Note that this calculation is based on the assumption that the different consumption groups are completed or abandoned independently of each other. If there are dependencies between different occurrences of a pattern and, hence, between the completion of different consumption groups, this can be incorporated in the probability calculation by using dependent / conditional probabilities. However, for the sake of simplicity of the presentation of technical concepts and algorithms, we use the formula for independent probabilities here.

```
1:  predictCompletionProbability (ConsumptionGroup CG) begin
2:   n ← Splitter.avgWindowSize − posInWindow
3:   if n ≤ 0 then
4:       n ← 1 // At least 1 more event expected
5:   end if
6:   T_n ← (1 − (n mod ℓ)/ℓ) * T_{⌊n/ℓ⌋*ℓ} + (n mod ℓ)/ℓ * T_{⌈n/ℓ⌉*ℓ}
7:   δ ← CG.completionState
8:   v_0 ← δ-th unit vector
9:   v_n ← T_n * v_0
10:  return v_n[last]
11: end function
```

Figure 5.5: Calculation of completion probability of a consumption group.

The dynamic process of pattern completion while processing events is modeled as a discrete-time *Markov process*. The state of the Markov process is spanned from δ to 0. For instance, if a pattern instance consists of at least 3 events (e.g., a sequence of 3 events, or a set of 3 events), the state-space has the elements "3", "2", "1" and "0", with "0" representing the state of total pattern completion. Based on statistics monitored at system run-time, a *stochastic matrix* T_1 is built that describes the *transition probabilities* between the states of the Markov process when processing *one* event. To this end, window versions of independent windows gather statistics about the probability of changing from δ_{old} to δ_{new} when an event is processed. The transition probabilities between any pair of δ_{old} and δ_{new} are captured in a matrix T_1^{new}. After ρ new measurements are available, an updated T_1 is computed from the old T_1^{old} and the newly calculated T_1^{new} as $T_1 = (1-\alpha) * T_1^{old} + \alpha * T_1^{new}$ (*exponential smoothing*). $\alpha \in [0,1]$ is a system parameter to control the impact of recent and of old statistics on T_1.

Now, the probability of state transitions when processing n events can be computed by raising T_1 to the n-th power: $T_n = (T_1)^n$. The initial state is modeled as a row vector $v_0 = (0,...,0,1,0,...,0)$—the δ-th unit vector, where the δ-th position is 1 and all other positions are 0. The probabilities of reaching the different states in n steps can be computed as $v_n = T_n * v_0$. The last entry of v_n, referring to state "0", is the probability to complete the pattern in n steps starting from state v_0.

To reduce the number of matrix multiplications, each time when T_1 is updated, a set of predefined "step sizes" is precomputed, e.g., T_{10}, T_{20}, T_{30}, etc., providing transition probabilities when 10, 20, 30, ... events are processed. If the number of expected events n is in between two precomputed steps, the transition probabilities are linearly

interpolated, e.g., $T_{14} = 0.6 * T_{10} + 0.4 * T_{20}$. The step size, denoted as ℓ, is a system parameter.

Figure 5.5 formalizes the described methods in an algorithm. The expected number of events left in the window, n, is calculated from the average window size monitored in the splitter and the position of the last processed event in the window (line 2). The probability matrix T_n is calculated by linear interpolation of precomputed matrices (line 6). δ is obtained directly from *CG* (line 7) and is used in order to build v_0 (line 8); v_n is calculated according to the description above (line 9). The resulting completion probability (transition to state " 0" / pattern completed) is returned (line 10).

Scheduling

Here, we describe how SPECTRE periodically selects and schedules the k window versions with the highest survival probability.

Notice that the survival probability of window versions is decreasing in a root-to-leaf direction in the dependency tree, i.e., in a window version's subtree there exist only window versions that have the same or a lower survival probability. Therefore, window versions are already sorted by their survival probability in the dependency tree, so that it already represents a max-heap, which simplifies the selection of the top-k versions substantially. From top to the bottom, window versions are added to the top-k list as detailed in the algorithm in Figure 5.6. The algorithm works with two data structures: (1) a set storing the resulting top-k versions (line 2), and (2) a priority queue storing candidates for being added to the top-k versions (line 3). The priority queue sorts the contained versions by their probability, highest probability first. Until k versions are found, the highest version from the candidate list is added to the result set (lines 4–6). The children of that version are also added as candidates (lines 7–9). This way, the top-k window versions are determined with only visiting the minimal number of vertices in the dependency tree.

The scheduling algorithm, listed in Figure 5.7, does not re-schedule window versions that are already scheduled to avoid unnecessary operations and to increase memory and cache locality of operator instances. Hence, the to-be-scheduled versions are determined (lines 7–9). Further, "free" operator instances are determined that will get a new window version scheduled (lines 10–11). Then, every window version that needs to be scheduled is scheduled to one of the free operator instances (lines 14–17).

```
1:  findTopKVersions (dependencyTree, k) begin
2:   result ← {} // set
3:   candidates ← {dependencyTree.root} // priority queue
4:   for i ← 1...k do
5:       tmp ← candidates.pop()
6:       result.append(tmp)
7:       for each M ← tmp.child do
8:           candidates.add(M)
9:       end for
10:  end for
11:  return result
12: end function
```

Figure 5.6: Top-k window version selection algorithm.

```
1:  List⟨OperatorInstance⟩ operatorInstances
2:  Tree dependencyTree
3:  schedule ( ) begin
4:   List⟨WindowVersion⟩ toBeScheduled // empty list
5:   List⟨OperatorInstance⟩ freeOperatorInstances ← operatorInstances
6:   List⟨WindowVersion⟩ topkVersions ← findTopKVersions(dependencyTree)
7:   for each WindowVersion WV in topkVersions do // first pass
8:       if not WV.isScheduled() then // WV must be scheduled
9:           toBeScheduled.add(WV)
10:      else// the operator instance keeps WV
11:          freeOperatorInstances.remove(WV.getOperatorInstance())
12:      end if
13:  end for
14:  for each WindowVersion WV in toBeScheduled do // second pass
15:      OperatorInstance OP ← freeOperatorInstances.pop()
16:      OP.scheduledWV ← WV
17:  end for
18: end function
```

Figure 5.7: Splitter: Scheduling algorithm.

5.2.3 Parallel Processing of Window Versions

Here, we describe how operator instances process their assigned window version according to the dependencies in the dependency tree. In particular, we describe how events are processed and suppressed, and how consumption groups are updated when sub-patterns are detected in a window version.

The scheduled window versions are processed in parallel by the associated operator instances. This means, that an operator instance processes or suppresses events according to the dependencies of the window version. In particular, when the root path of the window version meets the completion edge of a consumption group, events in that consumption group are not processed: they are suppressed. Complex events produced when processing a speculative window version are kept buffered until the window version either becomes valid—then, the complex events are emitted—or is dropped—then, the complex events are dropped, too. Further, when an event is processed, updates of the consumption groups can occur (creation, completion or abandoning a consumption group, or adding the event to an existing consumption group). In the following, we detail the underlying algorithms.

Figure 5.8 lists the algorithm for event processing in the operator instances. In the beginning of a processing cycle, the operator instance checks whether the splitter has scheduled a new window version (lines 7–9). Then, the next event of the currently scheduled window version is processed (lines 11–29). The operator instance checks whether the event is part of any consumption group that shall be suppressed (line 13). If this is the case, the event is suppressed, i.e., its processing is skipped. If the event is not suppressed, it is processed according to the operator logic (line 14). In doing so, there can be four different actions triggered based on feedback that the operator logic provides. (1) The processed event can complete one or multiple partial matches: This induces the creation of one or multiple complex events and the completion of the associated consumption groups. In that case, the emitted complex events are buffered, and the dependency tree is updated, calling the *consumptionGroupCompleted* function (cf. Section 5.2.1). (2) The processed event can lead to the abandoning of consumption groups, either by closing the window, or by invalidating the underlying partial match. In this case, the dependency tree is updated, calling the *consumptionGroupAbandoned* function (cf. Section 5.2.1). (3) The processed event can lead to the creation of a new consumption group by initiating a new partial match. In this case, the dependency tree is updated, calling the *consumptionGroupCreated* function (cf. Section 5.2.1). (4) The processed event can become part of one or several existing partial matches, possibly

```
 1: WindowVersion currentWV // currently processed WV
 2: WindowVersion scheduledWV // currently scheduled WV
 3: int i ← 0 // processing counter
 4:  main () begin
 5:   while true do
 6:       i ← i + 1
 7:       if scheduledWV ≠ currentWV then // changed WV?
 8:           currentWV ← scheduledWV
 9:       end if
10:
11:       // process the next event
12:       Event nextEvent ← currentWV.Window.getNextEvent()
13:       if nextEvent not in currentWV.suppressedCGs then
14:           Feedback fb ← process(nextEvent)
15:           if fb: emitted complex event E, completed CG_c then
16:               buffer E
17:               dependencyTree.consumptionGroupCompleted(CG_c)
18:           end if
19:           if fb: abandoned CG_a then
20:               dependencyTree.consumptionGroupAbandoned(CG_a)
21:           end if
22:           if fb: created CG_new then
23:               dependencyTree.
24:                  consumptionGroupCreated(CG_new, currentWV)
25:           end if
26:           if fb: added nextEvent to CG then
27:               CG.add(nextEvent)
28:           end if
29:       end if
30:
31:       // consistency check after each i steps
32:       if (i mod consistencyCheckFreq) == 0 then // consistency check
33:           bool inconsistencyDetected ← false
34:           for CG ∈ currentWV.suppressedCGs do
35:               if CG.version! = CG.lastCheckedVersion then
36:                   if currentWV.usedEvents ∩ CG.events ≠ ∅ then
37:                       inconsistencyDetected ← true
38:                   end if
39:               end if
40:               CG.lastCheckedVersion ← CG.version
41:           end for
42:           if inconsistencyDetected then
43:               rollback currentWV
44:           end if
45:       end if// end of consistency check
46:   end while
47:  end function
```

Figure 5.8: Operator Instances: Event Processing.

adding the event to the associated consumption groups. In this case, the affected consumption groups are updated directly without changing the structure of the dependency tree. Note, that in the implementation of SPECTRE, the function calls of the operator instances on the dependency tree are buffered—they are actually executed on the dependency tree in a batch at each new scheduling cycle of the splitter.

The k scheduled window versions are processed concurrently by the k operator instances without synchronizing the processing progress of the different window versions. This can lead to a situation where an update on an existing consumption group is propagated too late, causing inconsistencies. For instance, when an event is added to a consumption group CG in one window version WV_a after it has been processed in another window version WV_b adjacent to CG's completion edge, an inconsistency can be induced in WV_b (i.e., an event is processed that should be suppressed). To detect such situations, SPECTRE employs periodic *consistency checks*; the underlying algorithm is sketched in lines 31 – 45. For every consumption group to be suppressed in the currently processed window version, the algorithm checks whether an update has occurred since the last consistency check. If this is the case, the algorithm checks whether in the current window version, any event in the updated consumption group has been erroneously processed. If yes, then an inconsistency has been detected: The event should have been suppressed, but has actually been processed. If an inconsistency is detected, the state of the window version is rolled back to the start, i.e., the window version is reprocessed from the start. Instead of reprocessing a window version from the start in case of an inconsistency, it could also be recovered from an intermediate checkpoint. However, when implementing that approach, we realized that the overhead in periodically checkpointing all window versions is much higher than the gain from recovering from checkpoints.

5.3 Evaluations

In this section, we evaluate the performance of SPECTRE under different real-world and synthetic workloads and varying queries in the setting of an algorithmic trading scenario. We analyze the scalability of SPECTRE with a growing number of operator instances and the overhead involved in speculation and dependency management.

5.3.1 Experimental Setup

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

Evaluation Platform. We run SPECTRE on a shared memory multi-core machine with 2x10 CPU cores (Intel Xeon E5-2687WV3 3.1 GHz) that support hyper-threading (i.e., 40 hardware threads). The total available memory in the machine is 128 GB and the operating system is CentOS 7.3.

Implementation. SPECTRE is implemented using C++. The pattern detection and window splitting logic of the queries in these evaluations are implemented as a user-defined function (UDF) inside SPECTRE. Further, we provide a client program that reads events from a source file and sends them to SPECTRE over a TCP connection. Our implementation of SPECTRE is open source[2].

Datasets. We employ two different datasets centered around an algorithmic trading scenario.

First, a real-world stock quotes stream originating from the New York Stock Exchange (NYSE). This dataset contains real intra-day quotes of around 3000 stock symbols from NYSE collected over two months from Google Finance[3]; in total, it contains more than 24 million stock quotes. The quotes have a resolution of 1 quote per minute for each stock symbol. We refer to this dataset as the *NYSE Stock Quotes* dataset, denoted as *NYSE*. NYSE represents realistic data for stock market pattern analytics.

Second, we generated a random sequence of 3 million events consisting of 300 different stock symbols; the probability of each stock symbol is equally distributed in the sequence. We refer to this dataset as the *Random Stock Symbols* dataset, denoted as *RAND*. This dataset has been deliberately produced in a way such that the average completion probability of consumption groups is fixed. It is used for benchmarking the Markov model used in SPECTRE.

Queries. We employ three different queries, Q1 to Q3, in the evaluations (cf. Figure 5.9). The queries are listed in the extended `MATCH-RECOGNIZE` notation [ZWC07] introduced in Section 1.1.2. We further extended the `MATCH-RECOGNIZE` notation by a `CONSUME` clause (which stems from the TESLA event specification language [CM10]) to specify consumption policies.

[2]https://github.com/spectreCEP

[3]https://www.google.com/finance

[Q1]

PATTERN (MLE RE_1 RE_2 ... RE_q)
DEFINE
MLE **AS** (MLE.closePrice > MLE.openPrice),
RE_1 **AS** (RE_1.closePrice > RE_1.openPrice),
RE_2 **AS** (RE_2.closePrice > RE_2.openPrice),
...,
RE_q **AS** (RE_q.closePrice > RE_q.openPrice)
WITHIN *ws* events **FROM** MLE
CONSUME (MLE RE_1 RE_2 ... RE_q)

[Q3]

PATTERN (A SET(X_1 ... X_n))
WITHIN *ws* events
FROM every s events
CONSUME (A SET(X_1 ... X_n))

[Q2]

PATTERN (A B^+ C D^+ E F^+ G H^+ I J^+ K L^+ M)
DEFINE
A **AS** (A.closePrice < lowerLimit),
B **AS** (B.closePrice > lowerLimit **AND** B.closePrice < upperLimit),
C **AS** (C.closePrice > upperLimit),
D **AS** (D.closePrice > lowerLimit **AND** D.closePrice < upperLimit),
E **AS** (E.closePrice < lowerLimit),
F **AS** (F.closePrice > lowerLimit **AND** F.closePrice < upperLimit),
G **AS** (G.closePrice > upperLimit),
H **AS** (H.closePrice > lowerLimit **AND** H.closePrice < upperLimit),
I **AS** (G.closePrice < lowerLimit),
J **AS** (J.closePrice > lowerLimit **AND** J.closePrice < upperLimit),
K **AS** (G.closePrice > upperLimit),
L **AS** (L.closePrice > lowerLimit **AND** L.closePrice < upperLimit),
M **AS** (M.closePrice < lowerLimit),
WITHIN *ws* events **FROM** every s events
CONSUME (A B^+ C D^+ E F^+ G H^+ I J^+ K L^+ M)

Figure 5.9: Queries.

Q1 detects a complex event when the first q rising or the first q falling stock quotes of any stock symbol (defined as RE or FE, respectively) are detected within *ws* minutes from a rising or falling quote of a leading stock symbol (defined as MLE). The leading stock symbols are composed of a list of 16 technology blue chip companies. In the listing of Q1, we show only the stock rising pattern; the falling pattern is constructed accordingly. In case a complex event is detected, all constituent incoming events are consumed. Note, that this query always has a fixed pattern length of q, and each matching event moves the pattern detection to a higher completion stage.

Q2 is a query from related work (Balkesen and Tatbul [BDWT13], Query 9) that we extended by a window size of *ws* events, a window slide of *s* events and a consumption policy. It detects a complex event when specific changes occur in the price of a stock symbol between defined *upper* and *lower* limits. As in Q1, all constituent incoming events are consumed when a complex event is detected. We use the *lower* and *upper* limits to control the average pattern size. A small *lower* and a large *upper* limit results in a larger average pattern size, and vice versa. In contrast to Q1, Q2 has a variable length even for a fixed *lower* and *upper* limit. A matching event might or might not influence the pattern completion: the Kleene$^+$ implies that many events can match while the pattern completion does not progress.

Q3 detects a set of n specific stock symbols following stock symbol A. In contrast to the other queries, the ordering of those n symbols is not important. The pattern length n, window size ws, and window slide s can be freely varied. All constituent events are consumed when a complex event is detected.

5.3.2 Performance Evaluation

In this section, we evaluate the throughput and scalability of SPECTRE. First of all, we evaluate how SPECTRE performs with a growing number of parallel operator instances and with different consumption group completion probabilities. After that, we provide a detailed analysis of the Markov model SPECTRE uses to predict the completion probability of consumption groups. Finally, we discuss a comparison to the CEP engine T-REX [CM12a].

If not noted otherwise, we employ the following settings. The number of created consumption groups is limited to one per window version. The Markov model is employed with the parameters $\alpha = 0.7$ and $\ell = 10$.

To measure the system throughput, we streamed the datasets as fast as possible to the system. Each experiment was repeated 10 times. The figures show the 0th, 25th, 50th, 75th and 100th percentiles of the experiment results in a "candlesticks" representation.

Scalability

Here, we evaluate the scalability of SPECTRE. To this end, we analyze the system throughput, i.e., the number of events processed per second, with a growing number of operator instances. The following questions are addressed: (1) How does the scalability depend on the completion probability of the consumption groups? (2) How much computational and memory overhead is induced by maintaining the dependency tree and determining the top-k window versions?

Effect of Completion Probability of Consumption Groups We expect that the completion probability of consumption groups influences the system throughput. To make that clear, regard two extreme cases: All consumption groups are abandoned, or all consumption groups are completed. In the first case, SPECTRE should only schedule window versions on the *left-most* path of the dependency tree. In the second case,

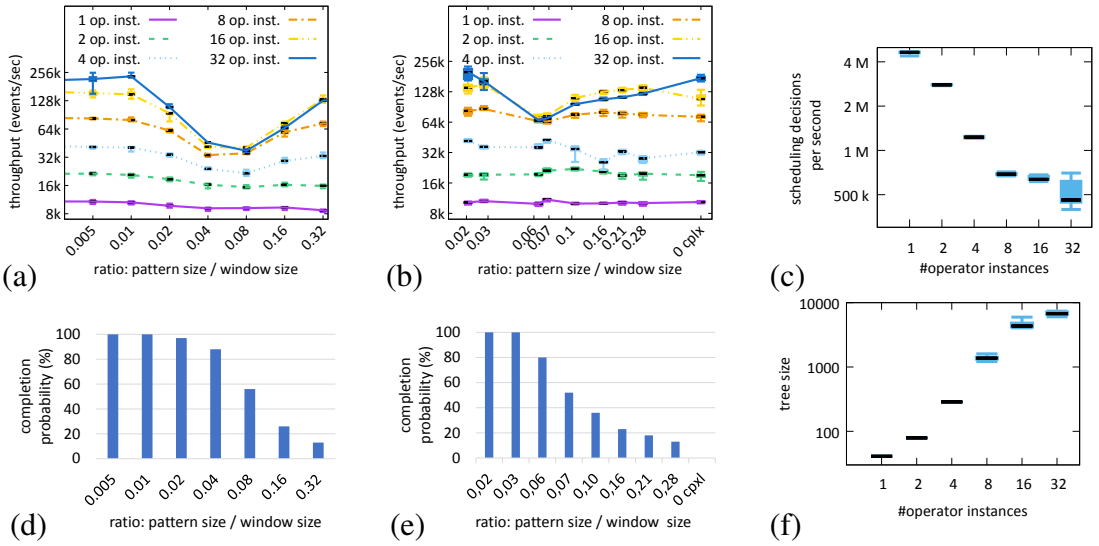


Figure 5.10: Evaluations. (a)+(d): Scalability (Q1 on NYSE). (b)+(e): Scalability (Q2 on NYSE). (c)+(f): Overhead (Q1 on NYSE).

SPECTRE should only schedule window versions on the *right-most* path of the dependency tree. In both cases, the scheduling algorithm should traverse the dependency tree in *depth*; i.e., it should schedule k window versions from k different windows. Further, none of the scheduled window versions should be dropped; all of them should survive. Hence, the throughput should be maximal. On the other hand, suppose that the completion probability of all consumption groups is constantly at 50 %. In that case, SPECTRE should traverse the dependency tree in *breadth*; i.e., it should schedule 1 window version of the first window, 2 window versions of the second window, 4 window versions of the third window, etc. However, only 1 window version of each window can survive; all others will be dropped. Hence, the higher k is, the more futile processing is performed, as the probability to predict the correct window version drops exponentially with k. In the following, we analyze whether SPECTRE shows the expected behavior and discuss implications.

To this end, we run a set of experiments with queries Q1 and Q2, using the NYSE dataset. In both queries, there are parameters that can be changed such that the average completion probability of consumption groups is manipulated. In Q1, we achieve this by directly setting the pattern size q, such that the ratio between pattern size and window size changes. Larger patterns are less likely to complete. In Q2, we cannot directly set the pattern size. However, we influence the average pattern size—and thus, the average completion probability—by changing the upper and lower limit parameters in the pattern definition.

In Q1, we employ a sliding window with a window size *ws* of 8,000 events, setting pattern sizes *q* of 40, 80, 160, 320, 640, 1280, and 2560 events. We calculate a "ground truth" value of the completion probability of consumption groups by performing a sequential pass without speculations: The number of created consumption groups divided by the number of produced complex events provides the ground truth value. The system throughput employing 1, 2, 4, 8, 16, and 32 operator instances, is depicted in Figure 5.10 (a). The corresponding ground truth probabilities are depicted in Figure 5.10 (d).

At a ratio of pattern size to window size of 40 / 8,000 (i.e., 0.005), the ground truth of consumption group completion probability is at 100 %, i.e., all partial matches are completed. The throughput scales almost linearly with a growing number of operator instances, from 10,800 events/second at 1 operator instance to 154,000 events/second at 16 operator instances (scaling factor 14.3) and 218,000 events/second at 32 operator instances (scaling factor 20.2). Increasing the pattern size decreases the completion probability of consumption groups. At a ratio of pattern size to window size of 640 / 8,000 (i.e., 0.08), the ground truth of consumption group completion probability is at 56 %, i.e., half of partial matches are completed and the other half are abandoned. The throughput scales from 9,200 events/second at 1 operator instance to 35,000 events/second at 8 operator instances (scaling factor 3.8). However, employing more than 8 operator instances does not increase the throughput further: With 16 and 32 operator instances, it is comparable to 8 operator instances. Further increasing the pattern size, we reach a ground truth of consumption group completion probability of 13 % at a ratio of pattern size to window size of 2560 / 8,000 (i.e., 0.32). Here, the throughput scales better, from 8,700 events/second at 1 operator instance to 131,900 events/second at 16 operator instances (scaling factor 15.2). Here, 32 operator instances do not improve the throughput further compared to 16 operator instances.

In Q2, we employ a sliding window with a window size *ws* of 8,000 events and a sliding factor *s* of 1,000 events. We arranged the lower and upper limit parameters in the pattern definition such that the corresponding average pattern sizes were 180, 226, 496, 560, 839, 1261, 1653, and 2223 events, plus one setting that made it impossible for a pattern to be completed. The system throughput employing 1, 2, 4, 8, 16, and 32 operator instances, is depicted in Figure 5.10 (b). The corresponding ground truth probabilities are depicted in Figure 5.10 (e).

At a ratio of pattern size to window size of 180 / 8,000 (i.e., 0.02), the ground truth of consumption group completion probability is at 100 %, i.e., all partial matches are

completed. The throughput scales almost linearly with a growing number of operator instances, from 10,300 events/second at 1 operator instance to 139,800 events/second at 16 operator instances (scaling factor 13.8) and 200,400 events/second at 32 operator instances (scaling factor 19.5). At a ratio of pattern size to window size of 560 / 8,000 (i.e., 0.07), the ground truth of consumption group completion probability is at 50 %, i.e., half of partial matches are completed and the other half are abandoned. The throughput scales from 10,900 events/second at 1 operator instance to 64,900 events/second at 8 operator instances (scaling factor 6.0). Employing more than 8 operator instances does not increase the throughput further: With 16 and 32 operator instances, it is comparable to 8 operator instances. When none of the partial matches can complete (denoted by "0 cplx"), the throughput scales from 10,400 events/second at 1 operator instance to 108,400 events/second at 16 operator instances (scaling factor 10.4) and 174,300 events/second at 32 operator instances (scaling factor 16.8).

Discussion of the results. We draw the following conclusions from the results. First of all, our assumptions on the system behavior are backed by the measurements. Further, the different queries impose "throughput profiles" that have a similar shape. The scaling behavior in SPECTRE, using the speculation approach, is very different from other event processing systems that have been analyzed in related work. In SPECTRE, the parallelization-to-throughput ratio largely depends on the completion probability of partial matches. This new factor leads to interesting implications when adapting the parallelization degree (i.e., elasticity), which is typically done based on event rates [DMM16, MKR15, LJK15] or CPU utilization [ABB+13, FMKP13]. Existing elasticity mechanisms do not take into account the completion probability to determine the optimal resource provisioning. Using the described throughput curves, SPECTRE could adapt the number of operator instances based on the current pattern completion probability.

Overhead of Speculation Here, we analyze the computational and memory overhead of maintaining the dependency tree in the splitter and scheduling the top-k window versions.

In a first experiment (Q1, NYSE dataset, q = 80, window size = 8,000), we measure how often the splitter can perform a complete cycle of tree maintenance and top-k scheduling per second. The cycle is described as follows: (a) Maintenance: performing all updates on the dependency tree that have been issued since the last maintenance, i.e., creating new consumption groups and window versions and delete dropped ones,

and (b) scheduling: schedule the new top-k window versions to the k operator instances according to the updated dependency tree.

In Figure 5.10 (c), the results are depicted. With 1 operator instance, SPECTRE achieves a maintenance and scheduling frequency of 4 million cycles per second. With increasing number of operator instances, the scheduling frequency decreases but is still considerably high, where SPECTRE achieves a scheduling frequency of $650,000$ and $450,000$ times per second with 16 and 32 operator instances, respectively. We conclude that there is some overhead involved in the management of the dependency tree and the scheduling algorithm, but there are no indications that this would become a bottleneck in the system.

Another concern about the dependency tree might be its growth and size in memory. To this end, we measured the maximal number of window versions maintained in the dependency tree at the same time (Q1, NYSE dataset, $q = 80$, window size = 8,000). The results of the experiments are depicted in Figure 5.10 (f). With 1 operator instance, the maximal tree size was at 41 window versions, growing up to 4,332 at 16 operator instances and 6,730 window versions at 32 operator instances. This is not a serious issue in terms of memory consumption. Indeed, the importance of a suitable top-k window version selection becomes obvious here: Determining the k window versions that will survive out of a large number of window versions that will eventually be dropped is a huge challenge, which SPECTRE could handle reasonably well in the performed experiments.

Markov Model

After we have discussed the overall system throughput and different factors that impact it, we go into a more detailed analysis of the completion probability model of consumption groups. In particular, we want to know how well the proposed Markov model behaves when the probabilities of complex events are changing. To this end, we perform two different experiments of query Q3 with different ratios of pattern size to window size: A ratio of 0.002 that has a high consumption group completion probability and a ratio of 0.1 that has a lower consumption group completion probability. We employed 32 operator instances and the window size ws was set to 1000 events where a new window is opened every 100 events ($s = 100$). We compare the proposed Markov model with a probability model that assigns each consumption group a fixed completion probability. The results of the two experiments are depicted in Figure 5.11 (a) and (b), respectively.

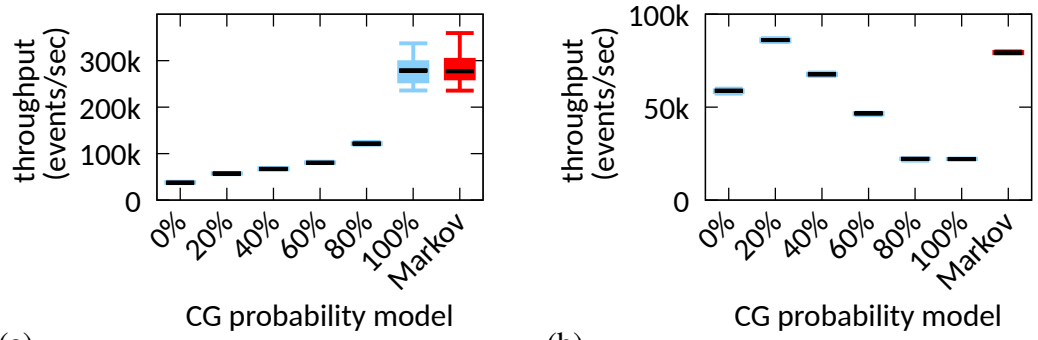


Figure 5.11: Evaluation of Markov Model.

At a ratio 0.002, the completion probability of a consumption group was at 100%. Accordingly, assigning a fixed probability of 100% to the consumption groups yielded a throughput of 279,000 events per second, which was significantly better than other fixed probabilities. The Markov model with a throughput of 277,000 events per second proved to be competitive with the best fixed model.

At a ratio of 0.1, the completion probability of a consumption group was at 32%. Accordingly, assigning a fixed probability of 20% to the consumption groups yielded a throughput of 86,000 events per second, which was significantly better than other fixed probabilities. The Markov model with a throughput of 79,000 events per second performed almost as good as the best fixed model.

From those results, we draw two conclusions. First, the Markov model is able to automatically learn suitable consumption group probabilities in different settings. Second, we can see that wrong probability predictions can cause a large throughput penalty.

Comparison to T-REX

We have also implemented query Q1 in the T-REX event processing engine [CM12a]. In total numbers, T-REX performed much worse than SPECTRE, reaching a throughput of only about 1,000 events per second. While this shows that the throughput of SPECTRE is competitive, it is worth to mention that both systems are different. T-REX is a general-purpose event processing engine that automatically translates queries into state machines, whereas SPECTRE employs user-defined functions to implement queries which allows for more code optimizations. T-REX does not support event consumptions in parallel processing, while SPECTRE can utilize multi-core machines to scale the throughput.

5.4 Related Work

In the past decades, a number of different CEP systems and pattern definition languages has been proposed. Besides CEP languages that do not support event consumptions, such as SASE [WDR06], the concept of event consumption gained growing importance. Based on practical use cases, Snoop [CM94] defined 4 different so-called *parameter contexts*, which are predefined combinations of selection and consumption policies. Building on a more systematic analysis of the problem, Zimmer and Unland [ZU99] proposed an event algebra that differentiated between 5 different selection and 3 different consumption policies that can be combined. Picking up and extending that work, the Amit system [AE04] allowed for distinct specifications of the selection and consumption policy. Finally, Tesla [CM10] and its implementation T-REX [CM12a] introduced a formal definition of its supported policies. The proposed speculation methods and the SPECTRE framework are applicable to any combination of selection and consumption policies.

Speculation has been widely applied to deal with out-of-order events in stream processing. Mutschler and Philippsen [MP14] propose an adaptive buffering mechanism to sort the events before processing them, introducing a *slack time*. When an event arrives outside of the slack time, results are recomputed. However, slack times cannot be used to overcome window dependencies in the event consumption problem: If one window is processed later, all depending windows would also need to be deferred. Brito et al. [BFSF08] as well as Wester et al. [WCN+09] propose transaction-based systems to roll-back processing when out-of-order events arrive. Their systems are not parallel, meaning that they only employ one speculation path for each operator. We also roll-back when window versions reach an inconsistent state. However, we propose a highly parallel multi-path speculation method (not only one path) and employ a probabilistic model to schedule the most promising window versions; hence, our system scales with an increasing number of CPU cores. Balazinska et al. [BKKL07] propose a system that quickly emits approximate results that are later refined when out-of-order events arrive. Our model would generally allow to be extended toward supporting probabilistic approximations, as a survival probability is given on the window versions. However, in this paper, we focus on consistent event detection (no false-positives, no false-negatives) and leave approximate applications of our model to the future work. Brito et al. [BFF09] propose for non-deterministic stream processing operators to mark events as speculative before logs have been committed to disc for consistent recovery. The speculative events can be forwarded to successor operators

in the operator graph that treat them specifically. In SPECTRE, speculative complex events are kept buffered until the window version is confirmed. We focus on providing deterministic event streams to the successor operators; in particular, we do not assume that subsequent operators or event sinks can handle events that are marked as speculative.

Alevizos et al. [AAP17] use Pattern Markov Chains to predict the completion probability and completion time of partial pattern matches. They use the prediction of pattern completion to trigger proactive actions, e.g., in the field of credit fraud detection. Their model assumes that the transition probabilities are stationary, whereas in SPECTRE, the transition matrix is updated frequently. Apart from that, their model is similar to our Markov model and was developed independently at the same time. Using the survival probabilities of window versions to trigger proactive actions can be an interesting further application of SPECTRE.

5.5 Conclusion

The SPECTRE system uses window-based data parallelization and optimized speculative execution of interdependent windows to scale the throughput of CEP operators that impose consumption policies. The novel speculation approach employs a probabilistic consumption model that allows for processing the k most promising window versions by k operator instances in parallel on a multi-core machine. Evaluations of the system show good scalability at a moderate overhead for speculation management.

6

Efficient Rollback-Recovery with Savepoints

We have discussed in Section 1.1.2 that in a distributed CEP system, operators hosted at potentially many different nodes of the network are taking a share in analyzing input streams and producing streams of outgoing events. Since many physical processes, e.g., the control of a manufacturing process, depend on the output of event processing systems, their correctness and performance characteristics are of critical importance. For CEP systems, this imposes strong requirements with respect to availability and consistency of their outgoing streams. In particular, the event streams provided to sinks of CEP systems should be indistinguishable from an execution in which the hosts of some operators fail or event streams are not available during a temporary partitioning of the network. The efficiency of reliable event processing can be measured with respect to its *runtime overhead* in a failure-free execution as well as its *recovery overhead* in the presence of failures.

In this chapter, a method is proposed for operator recovery that avoids any interruption of the CEP system and minimizes the amount of state to be transferred between nodes in a failure-free execution. The proposed approach relies on the observation that at certain points in time, the execution of a CEP operator solely depends on a distinct window of events from the incoming streams. So, the operator state only comprises necessary parts of the incoming streams and information about the current event window on them. Events from incoming streams can be reproduced from predecessor operators, so that only event sources need to provide outgoing streams in a reliable

way. However, information about the current event window is not reproducible and therefore is stored in a *savepoint* and replicated at other operators for fault tolerance.

This chapter includes preliminary work published in the diploma thesis of the author of this thesis [May12]. That work has been substantially extended and has been published in [KMR+13], which builds the basis of the main technical content of this chapter.

The contributions of this chapter comprise an expressive, general operator execution model that enables for any operator to externalize to an execution environment the current window of events it is processing. The model can be applied to all window-based operators. To illustrate its expressiveness, a comparison to the event specification language Snoop [CM94] is drawn. Based on this operator execution model, a *savepoint recovery system* is proposed that i) provides the basis in identifying an empty operator processing state, ii) manages the capturing and replication of savepoints and ensures the reproducibility of corresponding events, iii) implements a recovery in which also simultaneous failures of multiple operators can be tolerated.

6.1 System Model

The system and event processing model from Chapter 2 is extended as follows. For an event e in a stream (p,d) between two components p and d, we denote $SN(e)$ the sequence number of e in (p,d). $SN(e)$ is deterministically assigned by its producer; it is used in order to distinguish between different events in the same event stream that may have the same timestamp. We denote I_ω the set of incoming event streams at an operator ω, and O_ω the set of outgoing event streams.

In the previous chapters, we have assumed that operators implement a data parallelization architecture as described in Section 2.3. In this chapter, we first assume a sequential operator model, which is introduced in Section 6.2. Based on this model, we develop our operator recovery scheme. Later, in Section 6.7, we discuss how the recovery scheme can be implemented in the aforementioned data parallelization architecture.

The operators of the operator graph G are hosted by a set of n nodes, each node hosting possibly multiple operators. At any time, each node can fail according to the crash recovery model, where at most $k < n$ nodes are assumed to permanently fail or crash and recover an unbounded number of times. In addition, we will consider event sources and event sinks to be reliable. This means that each event produced by a source will

be accessible until all dependent operators have signaled that it can be discarded. Furthermore, event sources must be able to reliably store savepoints from their successors. Similarly, for events streamed to sinks we will use a fault tolerant delivery mechanism so that eventually a sink receives all events sent to it in the right order.

Note that we do not make any assumptions on timeliness for links connecting sources, operators and sinks, nor do we demand any synchronization of their clocks. The system can be realized as a highly distributed correlation network that involves communication over an Internet-like topology. We will use a monitoring component to suspect faulty processes and trigger reconfigurations of the placement of operators on nodes. The accurateness of this component, however, will only affect the performance, but not the correctness of the proposed method.

6.2 Approach Overview

Note that any approach that allows for the recovery of the state of failed operators requires the replication of state. The difference between different recovery methods is how and where state is replicated, e.g., at standby operators or at a persistent storage. One important observation from the rollback-recovery approach [EAWJ02] is that state $\Lambda(T)$ at a point in time T corresponds to state at a previous point in time T_{sp} plus a deviation $\Delta(\Lambda(T_{sp}), \Lambda(T))$ that happened on the state between T_{sp} and T. We are looking for the optimal T_{sp}, when the state of an operator is minimal, so that its replication requires only a minimum of resources.

Figure 6.1 shows a model of the components of an operator ω. Incoming events from I_ω are cached in a set of queues Q_I from which the *selector* determines windows of events to be mapped to outgoing events by the correlation function f_ω. Further, the selector can remove events from Q_I when they are no longer needed. The produced events are augmented with a sequence number by the sequencer and put into a set of queues Q_O from which they will be transferred to ω's successors in the operator graph.

The state $\Lambda(T)$ of ω comprises the states of Q_I, the selector, f_ω, the sequencer and Q_O. Observe that f_ω implements a mapping in its mathematical sense from a window w to sets of produced events, or, more precisely, attribute-value-pairs of events from w are mapped to attribute-value-pairs of produced events, each such mapping denoted a *correlation step*. Although f_ω builds up internal state, in this model there are no dependencies in between two subsequent correlation steps. Therefore, at a point in

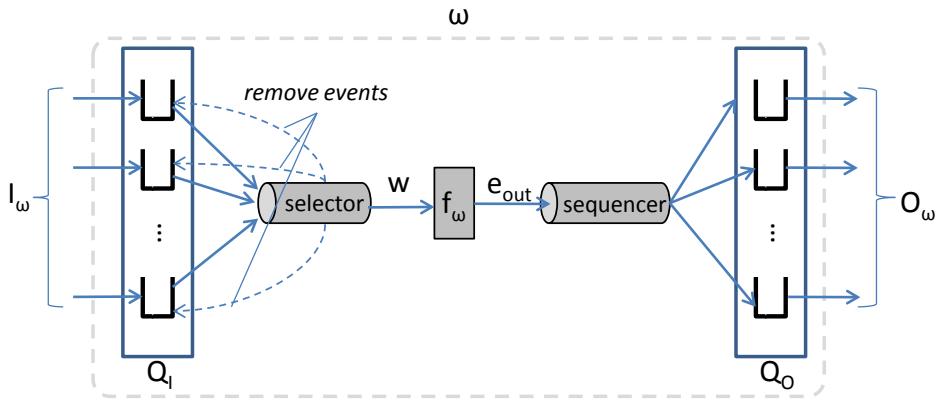


Figure 6.1: Operator Model.

time between two subsequent correlation steps, denoted T_{sp}, f_ω is *stateless*. The state of the sequencer only comprises one parameter, which is the next *SN* to be assigned to the next emitted event. In an arbitrary CEP operator, the state of the selector could be large, comprising manifold relations. For instance, subsequent windows could be inter-dependent because of intermediate event consumptions (cf. Chapter 5). In such a general case, taking a memory snapshot of the processing stack of the selector or implementing a custom state extraction method would be inevitable. However, the selector state can be drastically reduced with our operator execution model, introduced in Section 6.3, that prohibits inter-dependencies between subsequent windows. Indeed, as we show in Section 6.3.1, the state of the selector is reduced to one sequence number for each incoming event stream of the operator, pointing to the start events of a window. Finally, the state of Q_I and Q_O comprises all events contained in the operator's in and out queues. However, as we show in Section 6.5, in case of an operator failure, the state of Q_I can be reproduced by re-streaming events from the operator's predecessor, while Q_O can be reproduced by the operator.

In our *savepoint recovery system*, in order to be able to recover an operator ω once it has failed at a point in time T, the recovery procedure determines an earlier point in time $T_{sp} < T$ with the following properties: (i) f_ω is stateless, (ii) all events that Q_O contained at T_{sp} will never need to be reproduced by ω in the future, and (iii) events in Q_I at T_{sp} are available in ω's predecessors for re-streaming. Here, we outline how those properties are determined by the savepoint recovery system.

(i) Stateless times of f_ω are indicated to an execution environment by a hook that is installed in the correlation logic of the operators. The corresponding interface to the execution environment is specified in Section 6.3.

(ii) The second property is achieved by a distributed acknowledgment algorithm, which is described in Section 6.4.2. It synchronizes the points in time T_{sp} for recovery between adjacent operators. The general idea is that an *acknowledgment* or ACK indicates that an event does never have to be recovered any more, i.e., it has become "unnecessary". This has two consequences: First, such an event can be discarded from Q_O, as it will never have to be re-streamed to a successor of ω in the operator graph for recovery. Second, when recovering ω in case of a failure, it does not need to be able to re-produce the ACKed event any longer. This means that ω can move its recovery point T_{sp} to a point in time *after* the acknowledged event had been produced. The acknowledgment algorithm ensures that the latest recovery point T_{sp} at an arbitrary point in time T is always chosen with regard to this property.

(iii) The third property is also achieved by the acknowledgment algorithm (cf. Section 6.4.2), which only acknowledges events at a predecessor if they are not part of Q_I at T_{sp} anymore.

Now, the relevant *non-reproducible* state of ω at T_{sp} comprises only the state of the selector and the sequencer, which is captured in a *savepoint* and replicated at ω's predecessors. A savepoint only contains one sequence number for each incoming event stream (for the selector state) and one sequence number for the next event to be produced (for the sequencer state); hence, a savepoint is very lightweight. If ω fails at time T, $\Lambda(T_{sp})$ is restored from the replicated savepoint and from replayed events of Q_I from the predecessors. From this point on, the re-execution of ω, i.e., performing a sequence of correlation steps, will allow the operator to restore $\Lambda(T)$.

6.3 Execution Model

The following execution model refines the operator model introduced in Section 6.2. It describes the implementation of the selector and the sequencer, and defines the interface of an arbitrary operator implementation to these components. In doing so, we aim to keep the interface simple, so that existing implementations of f_ω can easily be embedded into the proposed system.

Let a window $\mathsf{w}\langle\langle SN_i^{start}, SN_i^{end}\rangle, \cdots\rangle$ on I_ω comprise for each incoming stream $i \in (in, \omega)$ all events between a start event with SN_i^{start} and an end event with SN_i^{end}. Then, w contains all incoming events on which f_ω executes one correlation step. Notice that, in contrast to the window model used in previous chapters, in this execution model, a

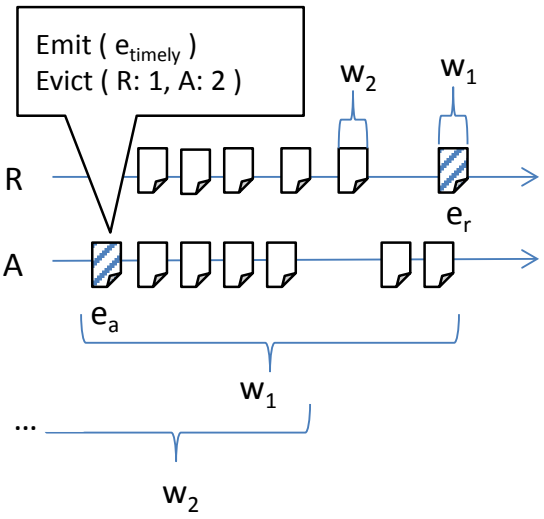


Figure 6.2: Example: Interface calls to the EE when shifting the window.

window has a distinct start and end event in *each* incoming event stream. This is useful in order to increase the expressiveness of our recovery scheme. In particular, certain combinations of selection and consumption policies, as defined in the *parameter contexts* of the Snoop pattern definition language [CM94], can be supported, which would not be possible with the simpler window model used in previous chapters.

To implement the correlation function f_ω, both the selector and the sequencer, which we together denote the *execution environment (EE)*, provide an interface. The interface of the selector is defined as EVICT(*removal*), with *removal* being a data structure that specifies how many events to remove from each incoming event stream in I_ω. The removal of n events from a stream $i \in (in, \omega)$ results in the slide of w to the next window w': $SN_i^{start'} = SN_i^{start} + n$, i.e., the start event moves further corresponding to the number of removed events. This is also known as *eviction* of events from the window [THS17]. Each correlation step has to result in at least one eviction in at least one of the incoming streams to ensure progress of the sliding window. That way, the selector keeps track of the start events of the windows, and just feeds events from Q_I to f_ω until its eviction interface is called and the next window starts. Events that are marked as evicted by f_ω get deleted from Q_I. The interface of the sequencer EMIT(*event*) takes a produced event from f_ω, assigns it a *SN* and puts it into Q_O.

Example (Figure 6.2): Consider a business monitoring system for which an operator ω monitors whether customer requests were successfully answered. In this scenario, ω is required to produce an alarm or confirmation depending on whether a customer request was answered within 10 minutes or not. The correct detection depends on the

successful detection of a customer request as well as detecting successful answers. In a correlation step, f_ω takes one event of type R (requests), say e_r, and then checks events of type A (answers) for a matching request ID attribute. The step ends when an event e_a of type A is reached that is either corresponding to e_r or has a timestamp that is more than 10 minutes older than the one of e_r, so that the 10 minutes timespan has been violated. In the first case, f_ω would emit an event e_{timely} and would evict e_r and events of type A that have a smaller timestamp than e_r, as answers are not expected to appear before the next request. In the second case, an event e_{alarm} is emitted and delivered to a special agent who will work on the request with a high priority. e_r and events of type A that have a smaller timestamp than e_r are evicted in this case, too. In the example in Figure 6.2, the window is shifted from w_1 to w_2 by evicting 1 event of type R and 2 events of type A.

6.3.1 Properties

Property 6.1 (STATE OF AN OPERATOR AT T_{sp}.) *Let T_{sp} be a point in time when an operator ω starts processing a new window w_{sp}. Then, $\Lambda(T_{sp})$ of ω comprises:*

- *Events in Q_I.*
- *The state of the selector: For each incoming stream $i \in (in, \omega)$: SN_i^{start} of w_{sp}.*
- *The state of the sequencer: The SN of the first event to be produced in w_{sp}.*

Explanation: Events in Q_I can be replayed from predecessors. In order to restore the selector, the *SN*s of the start events of w_{sp} have to be restored. Then, the selector can provide to f_ω exactly the same window that had been provided in the primary execution of the correlation step, which leads to the production of exactly the same events. The subsequent window w_{sp+1} depends only on w_{sp}, etc., so that all subsequent windows are indistinguishable from a failure-free execution. To restore the sequencer, it is initialized with the *SN* of the next event to be emitted.

Property 6.2 (START EVENTS OF CONSECUTIVE WINDOWS.) *For a window w_s, each start event has a higher or equal sequence number compared to the window w_p of a preceding correlation step.*

Explanation: Windows are moved when correlation steps are finished. Moving a window means that events are evicted; an eviction of an event from a window always leads to a higher sequence number of the next start event of the next window.

6.3.2 Expressiveness

After we have defined how the execution model of our event processing system works, here, we analyze its expressiveness in terms of support for CEP operator types. As a reference, we will take the event specification language *Snoop* [CM94] and analyze whether all event patterns that are formalized in Snoop can be implemented in our execution model. This has several reasons: First, Snoop has been motivated by real-world event processing scenarios. Such scenarios comprise, amongst others, sensor applications (e.g., hospital monitoring and global position tracking), applications that exhibit causal dependency (e.g., between aborts and rollbacks, bug reports and releases) and trend analysis and forecasting applications (e.g., security trading, stock market analysis). Second, Snoop is well-established in the scientific CEP community. But above all, Snoop provides a high expressiveness in comparison to other languages, as Cugola and Margara show in their article [CM12c].

In Snoop, complex event patterns can be correlated by *event operators*, which are the following ones: Disjunction, sequence, conjunction, aperiodic and periodic operators. Further, *parameter contexts* in Snoop describe the selection and consumption policy of events in a window. The following parameter contexts are defined: (i) *Recent*, where only the most recent occurrences of events of different type are selected. (ii) *Chronicle*, where incoming events are selected in the chronological order they occur. (iii) *Continuous*, where continuously each event that can possibly start a correlation is selected. (iv) *Cumulative*, where all events between a possible start and end event of a window are selected. In each of the parameter contexts except for *Continuous*, all selected events are consumed after correlation. For more detailed explanations and examples, please refer to the original Snoop paper [CM94].

Proposition 6.3 (EXPRESSIVENESS OF THE EXECUTION MODEL.) *All event operators and parameter contexts of Snoop can be implemented using the proposed execution model.*

PROOF In Snoop, *event expressions* define a finite time interval in which one or more atomic happenings, or events, can occur. Thus, they correspond to finite sets of event sequences. So they are equivalent to event windows as they are defined in the execution model. All Snoop event operators work solely with event expressions as operands. Thus, the sequence of execution iterations of Snoop event operators can be seen as an execution with a sequence of event windows $(w_1, w_2, w_3, ...)$ as operands.

Lemma 1 *For the operands* $(w_1, w_2, w_3, ...)$ *of a Snoop event operator, the following properties are satisfied:*

(i) $SN_j^{start} \in w_{k+1} \geq SN_j^{start} \in w_k \; \forall j \in (in, \omega)$,

(ii) For all event operators and parameter contexts, there exists an implementation of f_ω *so that when it is executed on* w_1, w_2, w_3..., *the events produced by* f_ω *satisfy the semantics of Snoop.*

PROOF (i) Proof by contradiction. If

$\exists\, e$ with $e \in w_{k+1} \wedge e \in j : SN(e) < SN_j^{start} \in w_k; j \in (in, \omega)$, then e would be in a later window but have a lower *SN* than the start event of w_k. This contradicts to the policy that sequence numbers are assigned sequentially on each event stream.

(ii) is satisfied because given the event windows, f_ω can implement any operations on a window, especially any functionality of a Snoop event operator can be implemented.■

Lemma 2 *None of the parameter contexts demands for the consumption of an intermediate event that is located within the window bounds of the next correlation step.* □

PROOF In *Recent*, when the detection of an event pattern in a window w_1 is completed, all events between the window start event and the event that completed the pattern are consumed. This cannot affect intermediate events of the next window w_2.

In *Chronicle*, events are selected in the order they occur in the window and then they are consumed, so no intermediate consumption can occur.

In *Continuous*, no explicit consumptions happen at all.

In *Cumulative*, all events of a window are consumed. ■

As a conclusion of Lemma 1 and Lemma 2, f_ω works on a sequence of windows that do not demand intermediate consumptions (i.e., consumptions of events in the middle of another window) and so it can implement the interface to the EE defined by the execution model. So, our execution model is at least as expressive as the Snoop event specification language. ■

6.4 Capturing and Replicating Savepoints

6.4.1 Log and Savepoint Management

Logs of Outgoing Events

Events in Q_I of ω are preserved in the Q_O of its upstream neighbors (i.e., predecessors), so that no additional events need to be transferred over the network at failure-free runtime. If ω fails, Q_I can be restored when its predecessors re-send their Q_O. Q_O must always contain enough events to restore the successor to its latest acknowledged state $\Lambda(T_{sp})$ which depends on the coordination of savepoints described in Section 6.4.2. Note, however, that outgoing events are reproduced when an operator recovers, so that events in Q_O are *reproducible* and do not need to be replicated.

Savepoints and Savepoint Trees

Savepoints contain the non-reproducible part of $\Lambda(T_{sp})$, which comprises the state of the selector and the sequencer. They are stored together with Q_O in the volatile memory of the predecessors of ω. If ω's predecessor is an event source, savepoints and events can be stored there in a reliable way, as event sources are assumed to survive failures.

So, when ω crashes at a point in time T, its predecessors hold all state information that is necessary to restore $\Lambda(T)$: $\Lambda(T_{sp})$ is restored from the savepoints and events from Q_O at predecessors, and $\Delta(T_{sp}, T)$ is restored by re-running ω from T_{sp} until T. We will determine later the points in time when an operator has to update and distribute its savepoint. To deal with asynchrony, it is necessary that all predecessors store a *complete* savepoint, so that at the recovery of ω a self-consistent savepoint is available, i.e., the information about the start events of w belong to the same w. In contrast to that, the retransmitted event streams in I_ω do not have to be consistent with regard to the same savepoint, because it is easily possible for a restored operator to discard events that stem from older windows and, hence, are not part of Q_I.

By now, only one failed operator can be restored at a time, but not several adjacent operators that fail at the same time. To make that possible, the non-reproducible part of $\Lambda(T_{sp})$, namely the savepoint, is replicated at all operators of the transitive closure of the predecessor relation in the operator graph, so that each operator preserves a *tree* of savepoints in its memory.

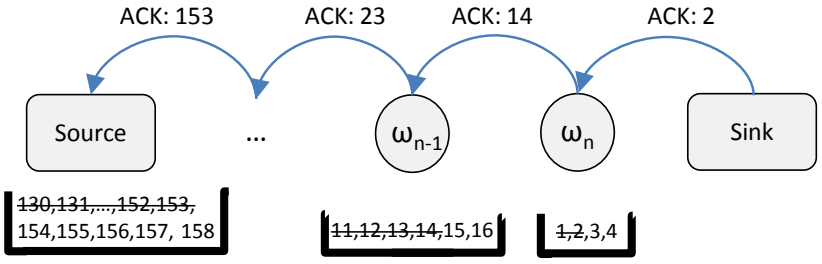


Figure 6.3: ACK flow and pruning of Q_O.

6.4.2 Coordination of Savepoints

In order to restore $\Lambda(T)$ of ω, the time of the latest savepoint T_{sp} to restore $\Lambda(T)$ has to be determined. This depends on the events that are part of Q_O at T; more precisely, the earlier the events of Q_O at T had been produced, the earlier is T_{sp}. A trivial implementation would never prune Q_O so that it contains all events an operator has ever produced. In case of recovery, T_{sp} would be "zero", i.e., the operator would be restored to the point in time when it initially started its work. To avoid that, it is necessary that an operator *prunes* Q_O from time to time, i.e., excludes events from $\Lambda(T)$ and increases T_{sp}. Events can be pruned when they are no longer necessary for the consistency of the event streams delivered to the event sinks. The necessity condition is defined as follows:

Definition 6.1 (Necessity of Events) *An event e is a necessary event if an event sink is interested in it and has not yet acknowledged its receipt.*

If all sinks interested in e have acknowledged its receipt at a point in time T_{ACK}, a predecessor operator ω can be sure that e is not necessary anymore and delete it (and all earlier events) from its Q_O. That way, e and all earlier events are not part of $\Lambda(T)$ for any $T \geq T_{ACK}$, so that T_{sp}, and hence, the savepoint, can be adjusted to the correlation step in which the first event following e had initially been produced. This is done by means of the *inverse correlation function* $f^{-1} : e \rightarrow \sigma_e$, which maps $SN(e)$ to the start events of the window w_e in which e had been produced. The most efficient implementation of f^{-1} is for each window to store SN_i^{start} for each stream $i \in (in, \omega)$ together with the produced events in Q_O. For each incoming stream, events before the corresponding start event of w_e are discarded from Q_I at T_{sp} and their SN is acknowledged at ω's predecessors. The predecessors proceed in the same way that ω does, prune their Q_O and adapt their T_{sp} to the production of events in Q_O, update their savepoint

accordingly in the savepoint tree, acknowledge *SN*s from f^{-1} at their predecessors and send them the updated savepoint tree for further replication.

Example (Figure 6.3): The sink in the depicted operator graph receives an event with $SN = 2$ from its predecessor ω_n. When the sink has received and processed that event in a way such that it does never need to receive that event again, it sends an ACK to ω_n containing $SN = 2$. Notice that the decision when to send such an ACK is up to the implementation of the application that received events from the CEP system. When receiving the ACK with $SN = 2$, ω_n prunes the events with $SN = 1$ and $SN = 2$ from its outgoing queue Q_O. Further, ω_n moves its own savepoint to a point in time after the event with $SN = 2$ has been produced, as in case of recovery of ω_n, it does not need to re-produce the event with $SN = 2$ (or any other earlier event). This *update* of the savepoint of ω_n results in a new ACK that ω_n send to its own predecessor ω_{n-1}. Suppose that events with $SN = 11$ through 14 from the incoming stream of ω_n where used by ω_n in order to produce the events with $SN = 1$ and 2 on its outgoing stream. Now that ω_n does not have to re-produce the events with $SN = 1$ and 2 on its outgoing stream any more (as they were ACKed), ω_n also does not have to receive the events with $SN = 11$ through 14 from the outgoing stream of its predecessor ω_{n-1} any more in case ω_n crashes and recovers. Thus, ω_n ACKs the events with $SN = 11$ through 14 at its predecessor ω_{n-1}.

As this example shows, to coordinate savepoints, we make use of ACKs which contain the *SN* of the acknowledged event. Additionally, the updated savepoint tree is piggy-backed on the ACK, if applicable. When receiving an ACK, an operator replaces the obsolete part in its savepoint tree, if applicable, prunes Q_O and checks whether T_{sp} can be updated. If this is the case, the operator sends an ACK to each of its predecessors. That way the ACKs flow *upstream*, i.e., against the flow direction of events, until they reach the event sources signaling that stored source events have become unnecessary and can be discarded.

The algorithm for log and savepoint maintenance of an operator is formalized in Figure 6.4. When an operator receives an ACK from one of its successors, it checks whether whether this ACK renders events from Q_O unnecessary (lines 6–7). If this the is the case, the operator proceeds as follows. To figure out which events in Q_I are rendered unnecessary, the inverse correlation function is evaluated (line 10), which returns the start events of the window in which the ACKed event had been produced. Events before those start events are pruned from Q_I (line 11). A new savepoint is created that captures the aforementioned window start events (line 12). Q_O is pruned,

```
1: Map<successor, ACK> latestRecACKs // Contains latest received ACK from each
   successor
2: Savepoint ownSP // Contains current own savepoint
3: List<Event> Q_I // Queues of incoming events
4: List<Event> Q_O // Queues of outgoing events

5: upon ⟨RECEIVEACK⟩(inACK)
6:  latestRecACKs.INSERT(inACK.producer, inACK)
7:  if latestRecACKs.GETOLDESTACKEDSEQ() has changed then // update own
   savepoint
8:      sn = inACK.GETACKEDSEQNO()
9:      e_acked = Q_O.GETEVENT(sn)
10:     map < instream, SeqNo > = f^-1(e_acked)
11:     Q_I.PRUNE // prune Q_I
12:     newSavepoint = NEW SAVEPOINT(map < instream, SeqNo >, sn)
13:     Q_O.PRUNE(sn) // prune Q_O
14:     SavepointTree = NEW SAVEPOINTTREE()
15:     SavepointTree.SETROOT(newSavepoint)
16:     for all ACK in latestRecAcks do
17:         SavepointTree.ADDCHILD(ACK.savepointTree)
18:     end for
19:     newACK = NEW ACK(SavepointTree)
20:     for all predecessors do
21:         SEND(newACK)
22:     end for
23:  end if
24: end
```

Figure 6.4: Algorithm for log and savepoint maintenance at an operator ω.

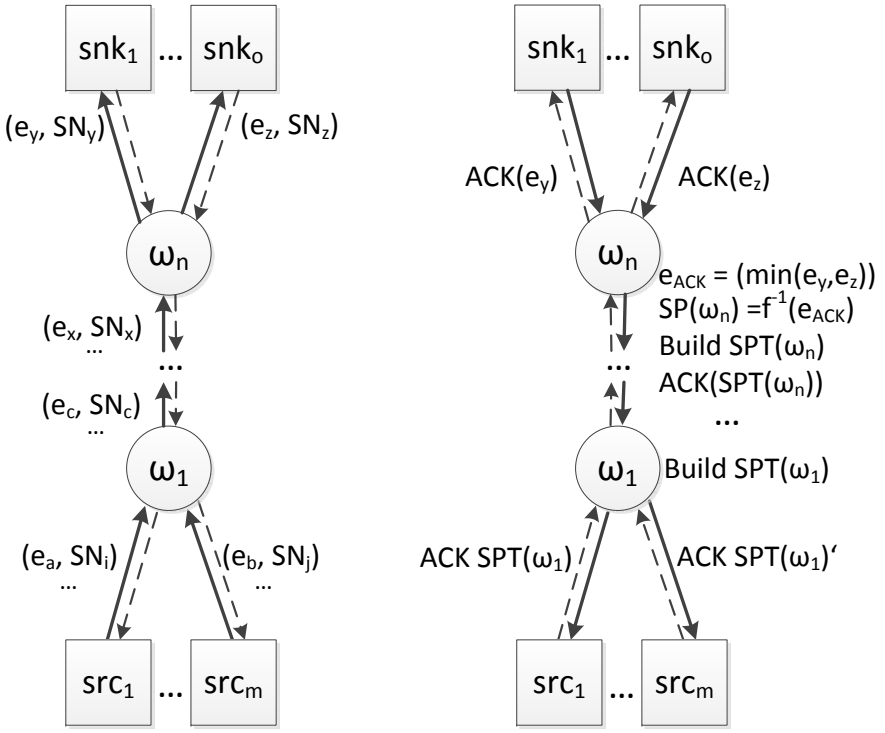


Figure 6.5: Example: Event and ACK flow in an operator graph.

such that it does not contain events older than the ACKed event (line 13). Now, the new savepoint tree of the operator is constructed (line 14). The new savepoint of the operator is the root of the savepoint tree (line 15). For each successor of the operator, the latest received savepoint tree of that successor is attached as a child to the root of the operator's savepoint tree (lines 16–18). The new savepoint tree is disseminated to all of the operator's predecessors (lines 19–22).

Figure 6.5 provides an example of the algorithm. In the left figure, events are flowing downstream, starting at the event sources and getting correlated with each other until some (complex) events are delivered to event sinks. In the right graph, the sinks acknowledge different *SN*s of received events. The minimal *SN* acknowledged by all connected event sinks at ω_n signals the latest unnecessary event. Accordingly, Q_O is pruned, the savepoint tree is updated and sent with an ACK to all predecessors. They store the new savepoint tree, update their own savepoint if applicable, send ACKs to their predecessors, etc., until finally the ACKs reach the event sources, where the savepoint trees are replicated and all acknowledged events are discarded.

```
1: upon ⟨RECEIVEINIT⟩(predecessors, successors)
2:   for all op in predecessors do
3:       SEND(op, RECOVERYREQUEST)
4:   end for
5:   while not received all RECOVERYINFORMATION do
6:       upon ⟨RECEIVERECOVERYNOTIFICATION⟩(predecessor)
7:         SEND(predecessor, RECOVERYREQUEST)
8:       end
9:       upon ⟨RECEIVERECOVERYINFORMATION⟩()
10:        list<RecoveryInformation>.ADD(RecoveryInformation)
11:      end
12:  end while
13:  RESTORESTATE(latestRecoveryInformation)
14: end

15: upon ⟨RECEIVERECOVERYREQUEST⟩(successor)
16:  SavepointTree = latestRecACKs.GET(successor).GETSAVEPOINTTREE()
17:  RecoveryInformation = NEW RECOVERYINFORMATION(Q_O, latestRecACKs)
18:  SEND(successor, RecoveryInformation)
19: end
```

Figure 6.6: Algorithms for recovery of an operator ω.

6.5 Algorithms for Operator Recovery

6.5.1 Recovery of the State of Failed Operators

For the description of the recovery algorithm, we will at first assume that operator failures are detected immediately and that failed operators are restarted automatically. Also, we assume that an operator knows his direct predecessors, even after it has crashed and recovered. From this point, we describe how an operator will be able to restore its state with regard to the latest available savepoint, so that event streams that were lost due to the failure get reproduced. Later, we will describe how the failure detection and operator topology management can be solved in an asynchronous system.

Recovery Procedure

The algorithms for the recovery of an operator are listed in Figure 6.6. After its restart, a failed operator ω sends a message to its predecessors that is called RECOVERYREQUEST (lines 2–4). When an operator receives such a RECOVERYREQUEST, it answers by sending the *recovery information* necessary for restoring the state of ω, which comprises Q_O (replay of the outgoing event stream) and the savepoint tree of ω (lines 15–19). ω waits until it has received all recovery information (lines 5–12). Then it identifies the answer among all answers from all its predecessors that contains the latest savepoint SP, which is the answer with the highest value for the *SN* of the next event to be produced (line 13). ω restores $\Lambda(T_{sp})$ by initializing the selector with the window defined in the SP, restores Q_I with the replayed events from the predecessors, and initializes the sequencer with the next *SN* to be assigned to a produced event.

To cope with multiple simultaneous failures of adjacent operators, ω sends a RECOVERYNOTIFICATION to its successors after its recovery. So, if one of those operators is awaiting recovery information from ω, it can detect that the RECOVERYREQUEST might have been lost because of a failure of ω and resend it. This way, a failed predecessor does not lead to an infinite waiting time of a restarted operator for receiving all recovery information. From bottom up, failed operators can recover, each sending the necessary recovery information to its successor, until all operators are restored to their latest ACKed state again.

6.5.2 Control and Adjustment of the Operator Topology

By now, we have assumed an error-free, immediate detection and restart of failed operators. However, in an asynchronous system, a perfect failure detector cannot be implemented to solve that problem. Instead, we have to work with a weaker failure detector abstraction that suspects operators to have failed, but the suspicions might be wrong.

Coordination of Operator Recovery

We employ a central component denoted *coordinator* which has global knowledge about the operator topology and is eventually always up and running, i.e., there might be times when the coordinator is not available, but it will always come back online.

The only state that needs to be recovered in case the coordinator fails is the knowledge of the operator topology, such that the availability of operators can be checked and recovery can be initiated if necessary. This state can be reliably stored in a redundant fashion, e.g., in a distributed data base. While the coordinator is not available, operator recovery would be delayed. In order to increase availability of the coordinator, it can be implemented in a distributed fashion. Section 6.7 discusses how this can be done.

The coordinator uses a failure detector with strong completeness (each failed operator will eventually be detected) and eventual weak accuracy (there is a time after which some correct process is never suspected), i.e., an *eventually strong failure detector* [CT96]. Such a failure detector checks for heartbeat messages that correct operators send in a certain frequency. If a heartbeat message from ω did not arrive at the coordinator within a *time bound* τ, it will be *suspected* to have failed. As we work with an asynchronous system model, the coordinator can never be sure whether the operator has really failed, but it is sure that a failed and not yet fully recovered operator will not send heartbeat messages anymore, so eventually every failure will be detected.

The algorithm at the coordinator for monitoring and control of the operators is listed in Figure 6.7. If ω is suspected to have failed, a replacement operator ω', i.e., an operator that implements the same correlation function as ω, is installed on a free system resource (lines 16–19). When ω' is initialized, it starts the recovery procedure described in Section 6.5.1. Now, it might be the case that the coordinator suspects ω' to have failed, too, so that ω'' is initialized, and so on. For that reason, suspected operators are not terminated immediately, but have the ability to run in parallel with their replacements. When the first of these parallel operators makes some real *progress* in event processing, the coordinator decides on that operator to remain in the topology and terminates all other replacement operators, i.e., they are shut down and their direct successors and predecessors are notified not to send messages to them any longer (lines 20–26). The notion of progress is defined as follows:

Definition 6.2 (PROCESSING PROGRESS OF AN OPERATOR.) *An operator ω has made progress after the restoration of its state when it updates its own savepoint for the first time.*

A savepoint update moves forward the point in time to which an operator gets recovered after its failure. That way, liveness of the system is guaranteed and the topology will finally stabilize. Note, that it is no problem for successors and predecessors of

```
 1: list<operator> operators // list of all operators
 2: list<operator> suspected // suspected operators
 3: list<operator> progressed // operators that have participated in the overall com-
    putational progress
 4: map<string, list<operator> > replacements // replacements of an operator

 5: procedure MONITORINGPROCEDURE()
 6:     while true do // infinite loop
 7:         nextCheck ← CURRENTTIME() + checkFrequency
 8:         for all operators do
 9:             CHECKLIVENESS(operator)
10:         end for
11:         wait until nextCheck
12:     end while
13: end procedure

14: procedure CHECKLIVENESS(operator)
15:     if (CURRENTTIME - lastReceivedHeartbeat.TIME ) > τ_operator then
16:         if operator ∉ suspected then
17:             suspected.ADD(operator)
18:             STARTREPLACEMENT(operator)
19:         end if
20:     else // operator is alive
21:         if operator ∈ suspected
22:             and operator ∈ progressed then
23:             RECALLREPLACEMENTS(operator)
24:             ADAPTTAU(operator, higher) // increase τ_operator
25:         end if
26:     end if
27: end procedure
```

Figure 6.7: Algorithm for monitoring and management of operator topology at the coordinator.

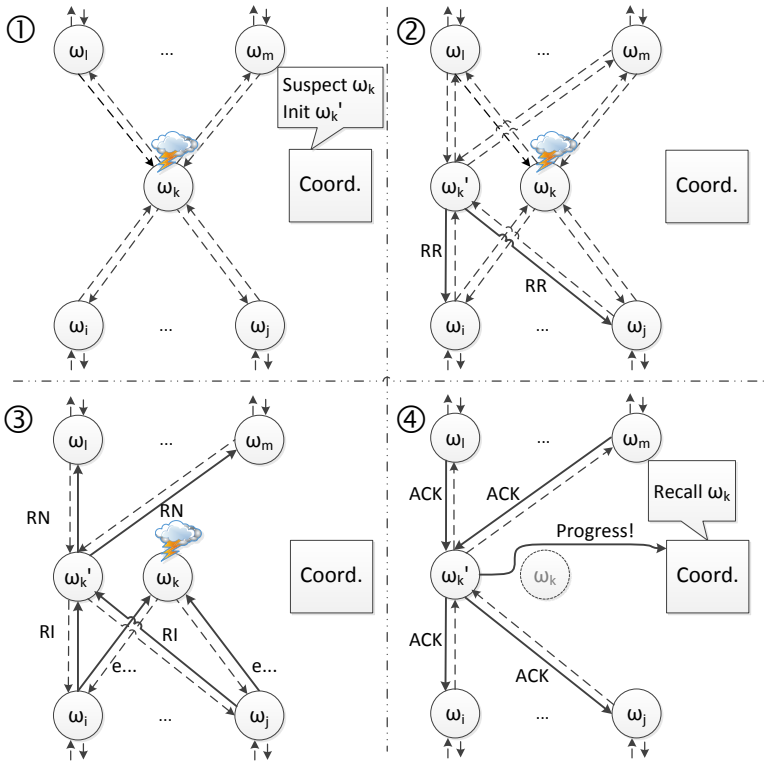


Figure 6.8: Recovery from an operator failure.

ω to cope with multiple replacements of ω running in parallel: As the replacements produce exactly the same events, the duplicates can easily be filtered, and ACKs are sent only to operators from which the ACKed events have been received.

When it turns out that ω had been suspected by mistake, i.e., when the coordinator had suspected ω and then received a heartbeat, the time bound τ can be adjusted to avoid such false suspicions in the future (Figure 6.7, line 24).

Example: Figure 6.8 shows how an operator ω_k fails and is replaced by ω'_k. The coordinator suspects ω_k and starts ω'_k. ω'_k initializes by sending RECOVERYREQUESTS to its predecessors, receives recovery information as responses, restores its state and sends RECOVERYNOTIFICATIONS to its successors. Now, ω'_k is fully incorporated into event production, and when enough events have been produced so that ACKs are received that make ω'_k update its savepoint, a progress notification is sent to the coordinator. Then, ω_k is deleted from the system, i.e., its successors and predecessors are notified to stop trying to communicate with ω_k.

6.5.3 Correctness Analysis

For proving completeness and consistency of event streams at the sinks, we prove that there are no false-negatives or false-positives and that the overall system makes processing progress despite an arbitrary number of simultaneous operator failures.

Proposition 6.4 (NO EVENT LOSS.) *In spite of the failure and recovery of an arbitrary number of operators at the same time, no necessary event in the sense of Definition 6.1 gets lost in an unrecoverable way.*

PROOF Let *snk* be a sink that has not yet received and acknowledged an event e_c. Let ω_{p1} be a direct predecessor of *snk*, ω_{p2} a direct predecessor of ω_{p1}, and so on. Then the latest savepoint of ω_{p1} is captured with respect to a point in time T_{sp_1} before e_c has been produced. So, e_c is reproducible by a recovered operator ω_{p1}. Further, the latest savepoint of ω_{p2} is captured with respect to a point in time when events that are part of $\Lambda(T_{sp_1})$ of ω_{p1} are reproducible, and so on, so that all necessary events are reproducible.

Proposition 6.5 (NO FALSE-POSITIVE EVENTS.)
Despite the simultaneous failure and recovery of an arbitrary number of operators, there are not delivered any events to the sinks that would not have been delivered in the failure-free execution of all operators.

PROOF Property 6.1 shows that the state of ω at T_{sp} contains exactly the information that is kept in a savepoint plus events from Q_I. As the savepoint is captured and replicated, it cannot deviate from the original savepoint after the recovery of ω. Further, necessary events from Q_I are either replayed from a predecessor or recursively reproduced, whereas the recursion stops at a point where events from some component of the operator graph are replayed (at the latest from the event sources). Events in Q_O are exactly the same events as originally sent in outgoing streams. As the recovered ω starts to process the same window on indistinguishable copies of the events from a failure-free operator execution, the produced events are indistinguishable, too.

Proposition 6.6 (LIVENESS OF THE SYSTEM.) *Events are delivered to the event sinks after a finite time from their physical occurrence, i.e., the CEP system makes progress in spite of the failure and recovery of an arbitrary number of operators.*

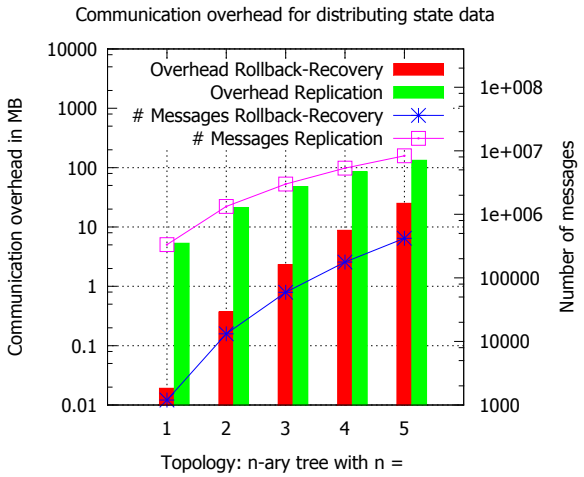


Figure 6.9: Run-time overhead comparison between rollback-recovery and active replication.

PROOF As only correct operators send heartbeat messages to the coordinator, failed operators will eventually be suspected and replacements are started. The topology stabilizes when an operator signaled processing progress with regard to Definition 6.2. So, the liveness of the system is ensured, given that only a finite number of hosts fails and there are enough correct hosts to run all operators. ■

6.6 Evaluation

In the evaluation, first of all we want to analyze the overhead of our approach induced at failure-free run-time: We measure the communication overhead and compare it with the overhead that would be induced by an active replication approach. Further, we analyze how the frequency of acknowledgments, the induced communication overhead and the size of Q_O are related. In doing so, we have implemented the algorithms in an event-based simulation without considering incidental influences like underlying hardware topologies and communication protocols in order to emphasize the *inherent overhead* that would be caused in any implementation on any underlying infrastructure. To this end, we employ an event-based simulation using the OMNeT++ simulation environment [VH08].

As a second aspect, we address the delay that the recovery of operators induces. In doing so, we identify significant parameters and develop a mathematical model of the recovery time of a failed operator.

6.6.1 Run-time Overhead

Our approach mainly induces run-time overhead with respect to two different aspects: The transmission of ACKs induces communication overhead, and the volatile storage of Q_O and savepoint trees impacts the memory footprint of operators and event sources.

Communication Overhead

The only data sent over the communication links at failure-free run-time are ACKs. We compute the size of an ACK as: $S(SimpleACK) + (\#Savepoints \times S(Savepoint))$. $S(SimpleACK)$ is 4 bytes for the acknowledged SN, $S(Savepoint)$ is 4 bytes for the SN of the next produced event, and $n \times 4$ bytes in a n-ary tree for the SNs of the start events in I_ω. An event is considered to be of a size of 16 bytes, 4 bytes for its SN, 4 bytes timestamp and 8 bytes payload. As the simulated operator topology, we chose n-ary trees with a depth of 3 for n = 1 to 5, with the root operator connected to 1 event sink and each leave operator connected to 1 event source. Event sources produce events with a frequency of 1 event / ms. We compare the overhead with the messaging overhead that would have been caused by duplicate events in the *active replication* approach [VKR11] developed by Völz et al. We assume a low replication factor of 2 and the best case scenario for the leader election (only one leader at a time), leading to an overhead that approximately equates to the number of events sent through the network regularly, neglecting the overhead that the leader election would cause. Figure 6.9 shows how much additional data is sent over the network within 5 minutes. As one can see, our rollback-recovery approach induces less communication overhead than the compared active replication approach. Conclusions on this are drawn in Section 6.6.3.

Memory Consumption

The consumption of main memory an operator or event source induces can be divided into two different parts: One part is the memory that is used for intermediate results in event processing, containing Q_I and the memory stack of f_ω. This part contains no specific overhead of the rollback-recovery approach, but rather the normal memory footprint of any event processing operator, so that we do not consider this in our evaluations. The other part is the memory used by all data stored solely for the purpose of enabling efficient rollback-recovery. This part is determined by two aspects: The size of the stored savepoint trees of the successors and the size of Q_O. The size

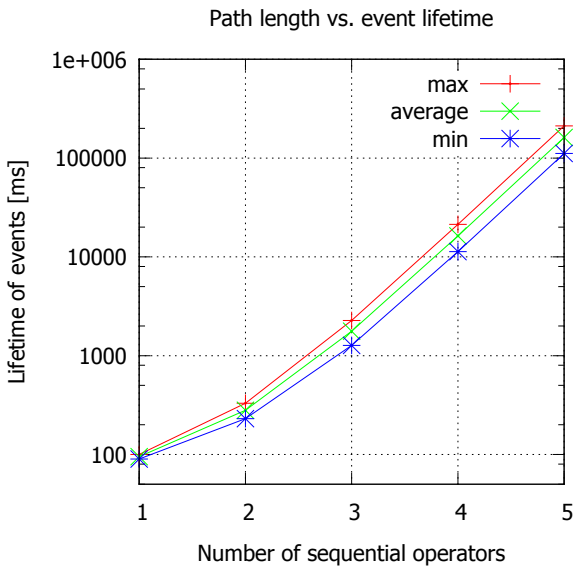


Figure 6.10: Lifetime of events in the sources against the number of sequential operators between sources and sinks.

of the savepoint trees depends solely on the operator topology and basically consists of one savepoint for each member of the transitive closure of the successor relation and is static (for a static operator topology). The size of Q_O, however, is dynamic and depends on the time between two consecutive ACKs that lead to its pruning.

To determine the maximal memory footprint, we analyze the event sources, as they have to store the maximal savepoint trees and events produced by sources have the maximal lifetime, i.e., the time between their production and storage in Q_O and the receiving of their ACK which triggers their deletion from Q_O. We have measured the influence of the complexity of the events delivered to the event sinks, i.e., the number of simple source events that are aggregated to a complex event, on the size of Q_O in the event sources. To do so, we built a simple topology containing one event source producing events in a frequency of 1 event / ms, a variable number of sequential operators and one event sink. In each correlation step, an operator takes 10 new events from its incoming stream and produces 1 outgoing event. Figure 6.10 shows the results: With an increasing complexity of the events delivered to the sink, the lifetime of events in the outlog of the event sources increases. Conclusions on this are drawn in Section 6.6.3.

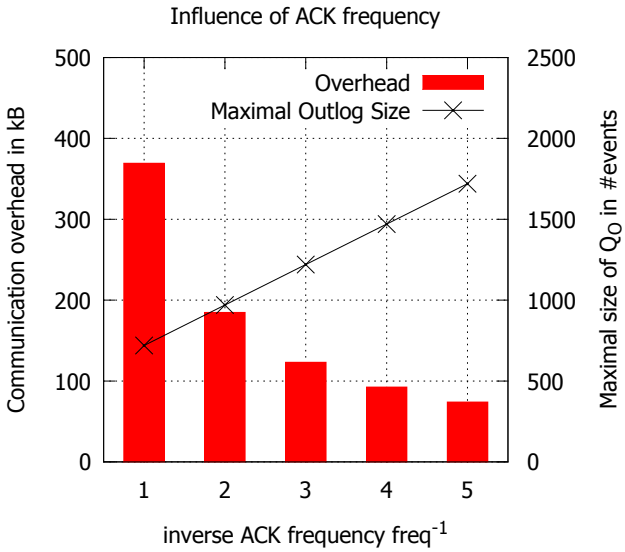


Figure 6.11: Influence of the ACK frequency at the event sink on the overall communication overhead and the maximal size of Q_O at the source.

Influence of ACK Frequency

We have further evaluated how the frequency of ACKs influences the maximal size of Q_O at sources and the run-time communication overhead. In doing so, we programmed the event sink to only acknowledge each *freq*-th event that it receives. The underlaying topology is a binary tree with a depth of 3, the rest of the parameters is as in the preceding scenarios. Figure 6.11 shows the results: When the ACK frequency decreases, the outlog size increases, but the communication overhead decreases. Conclusions on this are drawn in Section 6.6.3.

6.6.2 Analytical Model of Recovery Overhead

As the rollback procedures take some time until an operator state is restored, it takes longer for the system to recover from failures in comparison to active replication (where a replicated operator can take over processing with almost no latency). The recovery time of an operator ω is

$$recoverytime(\omega) = T_{fd} + T_{deploy} + T_{rec} + T_{pred}$$

with the parameters: (i) $T_{fd} = T_{channel\_delay} + T_{hb\_freq}$: Failure detection latency depends on the communication delay between operators and the coordinator and on the fre-

quency of heartbeat checks. (ii) T_{deploy}: Allocation of resources and deployment of the replacement operator mainly depends on the underlying technology such as communication channels and the availability of resources. For example, when using an elastic compute cloud such as Amazon EC2, it can take some minutes until a new node is allocated. The time can be reduced when pre-deployed operators are provided [FMKP13]. (iii) $T_{rec} = max(T_{channel\_delay}) + max(size(rec\_inf) \times channel\_rate)$: Recovery of the deployed replacement depends on the slowest connection to a predecessor and on the size of the recovery information. (iv) T_i: Recovery of predecessor operators in the case of i adjacent failures: $T_{pred} = \sum_{j=1}^{i-1} recoverytime(\omega_j)$. This is the sum of the recovery times of failed predecessors, which first need to be recovered successively in order to recover ω.

6.6.3 Conclusions on the Evaluation

Run-time Overhead

We see that in comparison to active replication the network load can be reduced drastically by applying the proposed approach. An increasing node degree causes that more sources participate in the production of simple events that get eventually aggregated to a complex event delivered to event sinks. That way, the number of simple events per source aggregated in such a complex event decreases and the frequency of ACKs increases. Therefore, the communication overhead increases faster with rollback-recovery than with active replication. However, this behavior can be controlled by the event sinks: If they decrease their frequency of ACKs, the overall network load decreases exponentially, but the size of Q_O at sources increases linearly. Besides the low network load, we do not need to preserve redundant resources as we would need to do in active replication. An additional advantage is that we are able to recover from multiple arbitrary operator failures with rollback-recovery (in fact, all operators can fail at the same time and be restored), a property which would cause immense costs in active replication.

Recovery Overhead

Recovery generally takes longer than in active replication. The main parameters highly depend on the communication channels and the provisioning of nodes to deploy re-

placement operators. The size of recovery information is limited to the size of the savepoint tree plus the amount of re-streamed events.

6.7 Extensions

This section discusses extensions of the proposed recovery method.

How can savepoint recovery be applied to the data parallelization framework introduced in Chapters 2 and 3?

Savepoint recovery builds upon the exposure of the internal window semantics and processing of the CEP operators to the recovery framework. When dealing with a parallel CEP operator that consists of a split–process–merge architecture, it is not straight forward to implement the interface to the execution environment. First of all, multiple operator instance process multiple windows in parallel. This raises the question of what the current savepoint of the operator is, and how it is determined and updated at run-time. Second, hosting different components of the parallel operator on different computing nodes yields the possibility of a *partial failure*, where, for instance, some operator instances fail, while others are still up and running.

Savepoint Management

The basic idea is that the splitter implements the interface to the execution environment of the savepoint recovery algorithm. This is intuitive, as the splitter already has the overview of all windows and their assignment to operator instances. Operator instances acknowledge each completely processed window to the splitter when the outgoing event is arrived and acknowledged at the merger. This way, the splitter knows the progress of processing of the windows.

Further, the splitter is also responsible for updating the operator savepoint. Hence, receiving ACKs from downstream operators, computing the new operator savepoint, and sending ACKs that contain the updated savepoint to the upstream operators, is all performed in the splitter. This means that in the savepoint protocol, other operators communicate with the parallel operator only via the splitter; the fact that the operator is a parallel operator is transparent to them.

Once the splitter gets an ACK from the successor operator, it checks whether to update the operator savepoint. The savepoint is updated with respect to the window the

acknowledged event was produced in, iff all previous windows also have been acknowledged by the operator instances. If not so, the savepoint cannot be updated yet, because some previous windows are still in operation in the operator instances.

Failure Recovery

When recovering from node failures, we differentiate the failure of the three different components of the parallelization framework: splitter, operator instances, and merger.

If the splitter fails, this is treated like a complete operator failure. This means that all operator instances and the merger are stopped and their state is released. Then, the splitter recovers according to the savepoint recovery protocol: the coordinator sends `RecoveryRequests` to the predecessors in the operator graph, and the splitter replica receives `RecoveryInformation` that contain the latest savepoints and event streams. When recovered, the splitter starts assigning windows to the operator instances, and the overall operator is recovered.

If one or more operator instances fail, the splitter sets up new operator instances (or uses the existing ones), assigning the unacknowledged windows that had been assigned to the failed operator instances to the running operator instances. Duplicate events that might exist in the merger are filtered.

If the merger fails, events emitted from the operator instances since the last savepoint's window need to be replayed to the merger to avoid event loss. Operator instances can buffer their complex events for restreaming. Hence, when the merger failure is detected, the splitter sets up a new merger and informs the operator instance to restream their outgoing event buffer to the recovering merger.

How can event consumptions be handled in savepoint recovery?

Event consumptions build up dependencies between different windows (cf. Chapter 5) and therefore are part of the operator state. Such consumptions need to be "replayed" when restoring an operator state. To make that possible, they can be stored in the operator savepoint, e.g., in a table that connects correlation steps with the performed event consumptions. Thus, the size of the savepoints and thereby the run-time overhead would increase.

How can the execution model by simplified for fixed window slide?

In some CEP operators, the window slide is fixed, e.g., by a fixed number of events or by a fixed time. Then, the execution environment can track the window movement without needing feedback from the operator about the number of evicted events. This

simplifies the interface between the EE and f_ω: f_ω just has to signal the end of each correlation step.

How can the coordinator be implemented in a distributed manner?

When distributing the coordinator functionality over different nodes, it is intuitive to make a predecessor the coordinator of a successor, as they communicate with each other anyway and heartbeat messages could be piggybacked. In doing so, it is important that each operator has exactly one coordinator responsible for its failure detection and recovery, so that the operator topology stabilizes. To solve this problem for asynchronous systems, concepts of leader election or group membership are necessary, e.g., as used in [FB98].

6.8 Related Work

The existing approaches to support fault tolerance for CEP and stream processing operators can be divided into three categories: The first category targets applications characterized as "partial fault-tolerant" [BBJ+, JSGAW09]. In the case of a failure, systems try to produce information which is not perfectly accurate but might still be useful to the receiver. In the second category, information is published tentatively and corrections can be issued at a later point in time that revoke the messages sent before [BFSF08,BFF09,HCCZ08,BBMS05]. These solutions are based on two premises: (i) Dependencies of operators on each other's output have to be within a reasonable limit to keep correction cost acceptable and, more important, (ii) the correction of incorrect messages has to be possible at all. In the scenarios we are examining, decisions might have already been made based on incorrect information that are either very costly or even impossible to correct. Therefore, accurate event streams, i.e., event streams without false positives or false negatives, are needed at the event sinks at all times.

Solutions that provide accurate event streams at all times involve the replication of functionality in active or passive replication [Sch90, BMST93], or rollback-recovery [EAWJ02] using checkpoints in combination with logs. Among others, these three principles have been applied to distributed stream processing systems; however, none of the current approaches provides all of the necessary properties for large-scale distributed CEP systems. In the following, we will discuss the proposed solutions individually.

Approaches using active or passive replication [HCZ08] typically incur quadratic overhead in terms of messages during failure-free execution. An optimized active replication approach proposed by Völz et al. [VKR11] employs a leader election algorithm to reduce the message overhead during failure-free runtime. Still, for tolerating f simultaneous failures, $f+1$ replicas are deployed for each operator, creating at least a linear message overhead. Martin et al. [MBF11] propose to use spare computing cycles that are available due to resource overprovisioning in elastic stream processing in order to perform the redundant computations of the replicas. This way, the additional resource cost of active replication can be reduced.

Rollback-recovery [EAWJ02], on the other hand, requires to take and store checkpoints at regular times. In doing so, checkpointing [KBG08, SM11] requires the execution of state-extraction algorithms that need either to be specified individually for each operator or require taking a full memory snapshot. Therefore, these approaches either restrict the user to using predefined operators only or require additional expertise to implement the extraction function on user-defined CEP operators. On the other hand, a memory snapshot can only be taken if the respective pages are write-locked, which incurs significant delay during failure-free operator execution.

In StreamMine3G, a stream processing system developed by Martin et al. [MSD+15], several fault tolerance schemes, such as active replication and rollback-recovery, are supported. A self-adaptive fault tolerance controller employs the most suitable fault tolerance scheme dynamically at run-time, taking into account the system workload and constraints provided by the user regarding recovery time and semantics. This way, the resource consumption of fault tolerance can be minimized, while the user's constraints are satisfied.

The basic idea to recompute outgoing event streams from incoming event streams originates in the "upstream backup" approach proposed by Hwang et al. [HBR+03, HBR+05]. In the terms of their work, in savepoint recovery, we target "deterministic operators", i.e., operators that always produce the same output when they receive the same incoming event streams. The guarantee we provide in savepoint recovery is of the type "repeating recovery", i.e., output tuples of a recovered operator are identical to those produced previously by the primary operator. Hwang et al. [HBR+03, HBR+05] in their work recognize that this would require a checkpointing mechanism in the operators. In savepoint recovery, we avoid heavy-weight checkpointing of arbitrary internal operator state by building the state extraction on top of the exposure of the operator's window semantics, so that operator state is only captured when it is minimal.

6.9 Conclusion

Although reliability is critical for many applications involving CEP systems, state-of-the-art approaches have clear shortcomings in providing accurate processing and low run-time overhead in a large-scale deployment.

This chapter proposed a novel rollback-recovery mechanism for multiple simultaneous operator failures in distributed CEP systems that eliminates the need for checkpoints and does not use persistent storage at operators. This way, it avoids the main drawbacks of previous approaches, which increase processing and network load for creating and maintaining large checkpoints, or burden application developers with defining operator specific mechanisms for checkpointing and recovery.

We defined an event processing model based on the concept of event windows to find points in time when a CEP system has minimal non-reproducible state which is then stored in replicated savepoints. The rest of the operator state can be reproduced from primary event streams, so that only event sources have to maintain events in a reliable way. An algorithm to coordinate savepoint maintenance over multiple levels of operators is provided, allowing to recover from simultaneous operator failures. We proved the algorithm correctness and provided evaluation results demonstrating its behavior in different parameter settings in comparison to active replication. The evaluations have shown that the network load can be reduced drastically, and that the frequency of acknowledgments at event sinks is a design parameter that can be used to balance between memory requirements and network load.

7

Conclusion and Outlook

This chapter closes the thesis by providing a summary of the presented results and an outlook to possible future research directions in this field.

7.1 Summary

The proliferation of sensors, e.g., in the Internet of Things, provides an unseen surge of streaming data being available for real-time data analytics. Often, situation-aware applications are interested in high-level situations in the surrounding world rather than in low-level sensor readings. To close the information gap between sensors and applications, the paradigm of Complex Event Processing (CEP) has been proposed [Luc01]. In a CEP system, a distributed network of operators step-wise detects patterns in event streams that correspond to the situations of interest. Nowadays, CEP systems are widely accepted both in academia as well as industry (Apache Flink [CKE+15], IBM Streams [IBM17], Oracle CEP [Ora17], Twitter Heron [KBF+15], WSO_2 Complex Event Processor [WS017], T-REX [CM12a], and many more). In CEP operators, to detect a search pattern, the infinite event sequence in the incoming event streams is restricted to a series of sub-sequences by means of a sliding window. In doing so, the window restricts the sequence of events that are allowed to build a search pattern. In many scenarios, such as traffic monitoring, social network analysis, and algorithmic trading, CEP operators face high and fluctuating workloads in their incoming event streams. Hence, it becomes necessary to enable CEP operators to exploit multi-core architectures and cloud computing by allowing for a distributed and parallel operator

execution. In doing so, data parallelization provides a high degree of parallelism by splitting the incoming event streams of an operator, processing the different partitions in parallel on an elastic set of identical operator copies, and merging the produced events into a deterministic order.

State of the art stream partitioning methods, i.e., key-based, batch-based and pane-based partitioning, are not suitable for all window-based operators. In particular, CEP operators often neither provide a key encoded in the events that is suitable for splitting (which excludes key-based partitioning), nor have fixed window sizes (which excludes batch-based partitioning), while temporal relations between the events prohibit splitting the windows into horizontal panes (which excludes pane-based partitioning). To overcome the shortcomings of existing methods, we have introduced a window-based data parallelization framework. It comes with an interface to expose the window semantics of the operators to the framework. The interface consists of two logical predicates, P_o and P_c, that evaluate when windows are opened and closed. This way, all window-based operators can be supported. The data parallelization framework comes with an elasticity controller, based on Queuing Theory, that adapts the parallelization degree, i.e., the number of operator instances, to the fluctuating event rates. The controller guarantees that a limit on events buffered in the operator is kept.

Scheduling subsequent overlapping windows to different operator instances in a Round-Robin fashion leads to good load balancing and low latency, but also induces a high network load between the splitter and the operator instances. This can be problematic in cloud environments, where multiple applications share the same data center network. Batch scheduling, i.e., scheduling multiple subsequent windows to the same operator instance in a batch, decreases the event replication and hence, the network load. However, it imposes temporary overload on the single operator instances, leading to latency spikes. To control the trade-off between communication overhead and latency, we have introduced a model-based controller that predicts the impact of batch scheduling a window on the latency in the operator instance. This way, the maximum amount of overlapping windows can be batched, while a latency bound is kept in the operator instances. We have shown that the communication overhead can be reduced by up to 76 % compared to Round-Robin scheduling.

When different overlapping windows are independent from each other, they can naturally be processed in parallel in different operator instances. However, consumption policies can impose dependencies between subsequent overlapping windows. A consumption policy may prohibit an event to be part of multiple search pattern instances.

That implies that the constituent events of a pattern instance detected in one window are excluded from all other windows as well, which breaks the data parallelism between different windows. To overcome the dependencies between windows, we proposed the SPECTRE framework for speculative processing of multiple dependent windows in parallel. In SPECTRE, based on the likelihood of an event's consumption in a window, subsequent windows may speculatively suppress that event. This way, we could reach an up to linear scalability of SPECTRE with the number of CPU cores, despite of window inter-dependencies.

When operators or computing nodes fail, internal operator state and events are lost. This usually leads to inconsistencies in the event streams delivered to the event sinks of the operator graph. To overcome this problem, operator state and events either have to be replicated, or recovered. Replication is expensive, as for each operator, f additional replicas have to be hosted in order to survive f operator failures. On the other hand, classical rollback-recovery requires to periodically take checkpoints of the operator state. This causes interruptions of the event processing, and demands for state externalization methods. To overcome the shortcomings of existing fault tolerance approaches, we proposed a novel method for rollback-recovery, called savepoint recovery. Our method allows for recovery from multiple simultaneous operator failures, but eliminates the need for persistent checkpoints. In savepoint recovery, the operator state is preserved in savepoints at points in time when current window of the operator is shifted. As in between two windows, no processing state is preserved, this reduces the size of the overall operator state to the current window position on the incoming event streams. The incoming event streams themselves are reproducible from upstream operators, so that a savepoint only needs to capture the window positions. We propose an expressive window-based event processing model to determine savepoints in an operator at run-time and algorithms for savepoint coordination in a distributed operator network. Evaluations show that very low overhead at failure-free run-time in comparison to other fault tolerance approaches is achieved.

7.2 Outlook

There are new technological trends that are likely to have a great impact on distributed, parallel CEP systems. Besides the trend of centralization of computing capabilities in cloud data centers, there is emerging a "counter movement", pushing computing resources to the edge of the network. The reason for this is intuitive: As the number

of data sources such as sensors is tremendously increasing, it is not feasible or economical anymore to ship *all* data to the cloud for analysis [BTMR17]. Instead, a more heterogeneous computing infrastructure needs to be developed and deployed. Here, we discuss new research challenges for CEP that emerge from that trend.

Fog Computing

Fog computing describes a computational continuum between cloud data centers and data sources and sinks at the edge of the Internet [BMZA12, Con17]. As such, fog computing is an important trend that facilitates the distribution of CEP operators. It consists of an orchestrated set of fog nodes that are typically hierarchically organized in multiple tiers [HLR+13, SHL+16, SGRM17]. Note, that fog computing is extending cloud computing beyond traditional cloud data centers that are just deployed closer to the network edge—referred to as edge clouds [TLG16]. Instead, fog computing features a great heterogeneity of the fog nodes; this exceeds the predefined flavors of virtual machines deployed on standard hardware connected by a data center network, as known in cloud computing. In particular, fog nodes may provide different types of CPUs and architectures (x86, ARM), specialized hardware (GPU, FPGA), and different types of persistent storage (HDD, SSD) and memory [Con17, MGSR17a, MGG+17, MGSR17b]. Moreover, network latency and bandwidth between different fog nodes becomes a larger issue compared to a single cloud data center.

When applying fog computing to distributed CEP systems, the heterogeneity of fog architectures, in comparison to cloud computing, poses new challenges on operator placement and elasticity. Early works in that field by Cardellini et al. try tackling the challenge by formulating the placement and parallelization problem as an integer linear program [CGLPN17]. Whilst this allows for computing the optimal replication degree and placement, the approach may not be scalable to a large number of operators and fog nodes. There is a need for efficient heuristics that solve the problem on the fly, as to account for the dynamics in distributed CEP systems and applications.

Another challenge in fog computing is that the domains of different fog layers may not be in control of a single administrative organization. This means that there might not be a global, fine-grained control of the end-to-end deployment of a CEP operator graph. As the field of fog computing is still in an early development stage, it is not clear yet how the interaction between different fog domains can be managed.

Load Shedding

In parallel CEP, a common assumption is the availability of unlimited computing resources in cloud environments. As a consequence, it is assumed to always be possible to increase the degree of operator parallelization by assigning additional computing resources. However, in many cases, computing resources are limited, e.g., if operators are executed on mobile nodes or in fog environments. Even in cloud environments, where computing resources in principle are unlimited, the available monetary budged might restrict the available computing power. Instead of increasing the parallelization degree, the quality of event detection in CEP can be reduced when the workload exceeds the computational capabilities of a CEP operator. This is usually achieved by skipping the processing of events or whole windows, i.e., load shedding.

Load shedding has been widely studied in stream processing, where aggregation of simple values is predominant [TcZ+03, TZ06, TcZ07, RBQ16]. In CEP pattern matching, the implications of load shedding to the quality of detection of complex patterns are not that well understood yet. Ottenwälder et al. [OKR+14b] proposed to use the metrics precision and recall in order to assess the quality of re-used results from similar ranges in answering moving range queries in mobility scenarios. This is for sure one possible metric for assessing load shedding strategies with regard to true/false positives/negatives ratios. However, other metrics might be more expressive in CEP scenarios. The metrics proposed by Zilberstein in the field of so-called “anytime algorithms” [Zil96], i.e., certainty, accuracy and specificity of the results, may lead toward more elaborate metrics suitable to CEP. Besides suitable quality metrics, further research questions are what load to shed in a single operator and where to shed load in the overall operator graph. In stream processing, typically, load shedding strategies regard operators as a black box, not using knowledge of the operator’s query or about internal state. The question is, how knowledge of the query of an operator can be exploited in load shedding, and whether the exposure of internal state, e.g., partial matches, of the operator can be exploited as well?

Bibliography

[AA13] Charu C Aggarwal and Tarek Abdelzaher. Social sensing. In *Managing and mining sensor data*, pages 237–297. Springer, 2013.

[AAP17] Elias Alevizos, Alexander Artikis, and George Paliouras. Event forecasting with pattern markov chains. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, DEBS '17, pages 146–157, New York, NY, USA, 2017. ACM.

[ABB+13] Tyler Akidau, Alex Balikov, Kaya Bekiroglu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, Daniel Mills, Paul Nordstrom, and Sam Whittle. MillWheel: Fault-tolerant Stream Processing at Internet Scale. *Proc. VLDB Endow.*, 6(11):1033–1044, August 2013.

[ABW06] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[AC11] Raman Adaikkalavan and Sharma 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*, DASFAA'11, pages 341–356, Berlin, Heidelberg, 2011. Springer-Verlag.

[AE04] Asaf Adi and Opher Etzion. Amit - the situation manager. *The VLDB Journal*, 13(2):177–203, May 2004.

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Comput. Netw.*, 54(15):2787–2805, October 2010.

[AXV+11] Apoorv Agarwal, Boyi Xie, Ilia Vovsha, Owen Rambow, and Rebecca Passonneau. Sentiment analysis of twitter data. In *Proceedings of the Workshop on Languages in Social Media*, LSM '11, pages 30–38, Stroudsburg, PA, USA, 2011. Association for Computational Linguistics.

[BAJR14] Pramod Bhatotia, Umut A. Acar, Flavio P. Junqueira, and Rodrigo Rodrigues. Slider: Incremental sliding window analytics. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 61–72, New York, NY, USA, 2014. ACM.

[BBD+04] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Dilys Thomas. Operator scheduling in data stream systems. *The VLDB Journal*, 13(4):333–353, December 2004.

[BBJ+] Nikhil Bansal, Ranjita Bhagwan, Navendu Jain, Yoonho Park, Deepak Turaga, and Chitra Venkatramani. Towards Optimal Resource Allocation in Partial-Fault Tolerant Applications. In *Proceedings of the 27th IEEE Conference on Computer Communications, INFOCOM '08*, pages 1319 –1327.

[BBMS05] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 13–24, New York, NY, USA, 2005. ACM.

[BCKR11] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards predictable datacenter networks. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 242–253, New York, NY, USA, 2011. ACM.

[BDWT13] Cagri Balkesen, Nihal Dindar, Matthias Wetter, and Nesime 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*, DEBS '13, pages 3–14, New York, NY, USA, 2013. ACM.

[BEH+10] Dominic Battré, Stephan Ewen, Fabian Hueske, Odej Kao, Volker Markl, and Daniel Warneke. Nephele/pacts: a programming model and execution framework for web-scale analytical processing. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 119–130, New York, NY, USA, 2010. ACM.

[BFF09] Andrey Brito, Christof Fetzer, and Pascal Felber. Minimizing latency in fault-tolerant distributed stream processing systems. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 173–182, June 2009.

[BFSF08] Andrey Brito, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. Speculative out-of-order event processing with software transaction memory. In *Proceedings of the Second International Conference on Distributed Event-based Systems*, DEBS '08, pages 265–275, New York, NY, USA, 2008. ACM.

[BKKL07] Magdalena Balazinska, YongChul Kwon, Nathan Kuchta, and Dennis Lee. Moirae: History-enhanced monitoring. In *CIDR*, pages 375–386. Citeseer, 2007.

[BMK+11] Andrey Brito, André Martin, Thomas Knauth, Stephan Creutz, Diogo Becker, Stefan Weigert, and Christof 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*, CloudCom '11, pages 48–58, 2011.

[BMST93] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Distributed systems (2nd ed.). chapter "The primary-backup approach", pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[BMZA12] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC '12, pages 13–16, New York, NY, USA, 2012. ACM.

[BOO09] IT-Information Technology. volume 51, page 80. Oldenbourg Verlag, October 2009.

[Bos14] Sanjay Kumar Bose. M/m/m/inf queue, 2014. http://nptel.ac.in/courses/117103017/6.

[BT11] Cagri Balkesen and Nesime Tatbul. Scalable data partitioning techniques for parallel sliding window processing over data streams. International Workshop on Data Management for Sensor Networks (DMSN), 2011.

[BTMR17] Thomas Bach, Muhammad Adnan Tariq, Ruben Mayer, and Kurt Rothermel. Knowledge is at the edge! how to search in distributed machine learning models. In *On the Move to Meaningful Internet Systems. OTM 2017 Conferences: Confederated International Conferences: CoopIS, C&TC, and ODBASE 2017, Rhodes, Greece, October 23-27, 2017, Proceedings, Part I*, pages 410–428, Cham, 2017. Springer International Publishing.

[BTO13] Cagri Balkesen, Nesime Tatbul, and M. Tamer Özsu. Adaptive input admission and management for parallel stream processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 15–26, New York, NY, USA, 2013. ACM.

[BYV+09] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging it platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599 – 616, 2009.

[CBB+03] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Donald Carney, Ugur Cetintemel, Ying Xing, and Stan Zdonik. Scalable Distributed Stream Processing. In *CIDR 2003 - First Biennial Conference on Innovative Data Systems Research*, Asilomar, CA, January 2003.

[CCA+10] Tyson Condie, Neil Conway, Peter Alvaro, Joseph M. Hellerstein, Khaled Elmeleegy, and Russell Sears. Mapreduce online. In *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association.

[CcR+03] Don Carney, Uğur Çetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Mike Stonebraker. Operator scheduling in a data stream manager. VLDB '03, pages 838–849. VLDB Endowment, 2003.

[CGB+11] Badrish Chandramouli, Jonathan Goldstein, Roger Barga, Mirek Riedewald, and Ivo Santos. Accurate latency estimation in a distributed event processing system. In *2011 IEEE 27th International Conference on Data Engineering*, pages 255–266, April 2011.

[CGLPN16] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Optimal operator placement for distributed stream processing applications. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, pages 69–80, New York, NY, USA, 2016. ACM.

[CGLPN17] Valeria Cardellini, Vincenzo Grassi, Francesco Lo Presti, and Matteo Nardelli. Optimal operator replication and placement for distributed stream processing systems. *SIGMETRICS Perform. Eval. Rev.*, 44(4):11–22, May 2017.

[CJ09] Sharma Chakravarthy and Qingchun Jiang. Load shedding in data stream management systems. In *Stream Data Processing: A Quality of Service Perspective*, volume 36 of *Advances in Database Systems*, pages 137–166. Springer US, 2009.

[CKE+15] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache flink: Stream and batch processing in a single engine. *IEEE Data Eng. Bull.*, 38:28–38, 2015.

[CM94] Sharma Chakravarthy and Deepak Mishra. Snoop: An expressive event specification language for active databases. *Data & Knowledge Engineering*, 14(1):1 – 26, 1994.

[CM10] Gianpaolo Cugola and Alessandro Margara. Tesla: A formally defined event specification language. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 50–61, New York, NY, USA, 2010. ACM.

[CM12a] Gianpaolo Cugola and Alessandro Margara. Complex event processing with t-rex. *Journal of Systems and Software*, 85(8):1709 – 1728, 2012.

[CM12b] Gianpaolo Cugola and Alessandro Margara. Low latency complex event processing on parallel hardware. *Journal of Parallel and Distributed Computing*, 72(2):205 – 218, 2012.

[CM12c] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys*, 44(3):1–62, June 2012.

[Con17] OpenFog Consortium. OpenFog Reference Architecture. `https://www.openfogconsortium.org/ra/`, 2017. [Online; accessed 17-Feb-2017].

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996.

[DMM16] Tiziano De Matteis and Gabriele Mencagli. Keep calm and react with foresight: Strategies for low-latency and energy-efficient elastic data stream processing. In *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '16, pages 13:1–13:12, New York, NY, USA, 2016. ACM.

[DMM17a] Tiziano De Matteis and Gabriele Mencagli. Elastic scaling for distributed latency-sensitive data stream operators. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 61–68, March 2017.

[DMM17b] Tiziano De Matteis and Gabriele Mencagli. Parallel patterns for window-based stateful operators on data streams: An algorithmic skeleton approach. *International Journal of Parallel Programming*, 45(2):382–401, Apr 2017.

[DZSS14] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. Adaptive stream processing using dynamic batch sizing. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 16:1–16:13, New York, NY, USA, 2014. ACM.

[EAWJ02] E. N. (Mootaz) Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys*, 34:375–408, September 2002.

[FB98] Massimo Franceschetti and Jehoshua Bruck. A Leader Election Protocol for Fault Recovery in Asynchronous Fully-Connected Networks. Technical report, California Institute of Technology, 1998.

[Fei15] Dror G. Feitelson. *Workload Modeling for Computer Systems Performance Evaluation*. Cambridge University Press, New York, NY, USA, 1st edition, 2015.

[FHC12] Todd Michael Franke, Timothy Ho, and Christina A. Christie. The chi-square test: Often used and more often misinterpreted. *American Journal of Evaluation*, 33(3):448–458, 2012.

[FMKP13] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter 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*, SIGMOD '13, pages 725–736, New York, NY, USA, 2013. ACM.

[For17] Forbes. Internet Of Things Market To Reach $267B By 2020. `https://www.forbes.com/sites/louiscolumbus/2017/01/29/internet-of-things-market-to-reach-267b-by-2020/#3d59aee6609b`, 2017. [Online; accessed 26-Jun-2017].

[Ged14] Buğra Gedik. Partitioning functions for stateful data parallelism in stream processing. *The VLDB Journal*, 23(4):517–539, 2014.

[GHJ+09] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 51–62, New York, NY, USA, 2009. ACM.

[GJK+17] Vincenzo Gulisano, Zbigniew Jerzak, Roman Katerinenko, Martin Strohbach, and Holger Ziekow. The debs 2017 grand challenge. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, DEBS '17, pages 271–273, New York, NY, USA, 2017. ACM.

[GJPPnM+12] Vincenzo Gulisano, Ricardo Jiménez-Peris, Marta Patiño Martínez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23(12):2351–2365, Dec 2012.

[GMM+16] Michael Grossniklaus, David Maier, James Miller, Sharmadha Moorthy, and Kristin Tufte. Frames: Data-driven windows. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, pages 13–24, New York, NY, USA, 2016. ACM.

[GSHW14] Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. Elastic scaling for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 25(6):1447–1463, June 2014.

[GSTH11] Donald Gross, John F. Shortle, James M. Thompson, and Carl M. Harris. *Fundamentals of Queueing Theory*. Wiley Series in Probability and Statistics. Wiley, 2011.

[Gue12] Vincenzo Guerriero. Power law distribution: Method of multi-scale inferential statistics. *Journal of Modern Mathematics Frontier (JMMF)*, 1:21–28, 2012.

[GYX+15] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, Zhi-Wei Lin, and Varugis Kurien. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 139–152, New York, NY, USA, 2015. ACM.

[Hay08] Brian Hayes. Cloud computing. *Commun. ACM*, 51(7):9–11, July 2008.

[HBR+03] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. A Comparison of Stream-Oriented High-Availability Algorithms. Technical Report CS-03-17, Brown University, September 2003.

[HBR+05] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Ugur Cetintemel, Michael Stonebraker, and Stan Zdonik. High-availability

algorithms for distributed stream processing. In *Proceedings of the 21st IEEE International Conference on Data Engineering (ICDE'05)*, pages 779–790, April 2005.

[HCCZ08] Jeong-Hyon Hwang, Sanghoon Cha, Uğur Cetintemel, and Stan Zdonik. Borealis-r: A replication-transparent stream processing system for wide-area monitoring applications. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 1303–1306, New York, NY, USA, 2008. ACM.

[HCZ08] Jeong-Hyon Hwang, Ugur Cetintemel, and Stan Zdonik. Fast and Highly-Available Stream Processing over Wide Area Networks. In *Proceedings of the IEEE 24th International Conference on Data Engineering, ICDE '08*, pages 804 –813, 2008.

[HHKA14] Nikolas Roman Herbst, Nikolaus Huber, Samuel Kounev, and Erich Amrehn. Self-adaptive workload classification and forecasting for proactive resource provisioning. *Concurrency and Computation: Practice and Experience*, 26(12):2053–2078, 2014.

[Hir12] Martin Hirzel. Partition and compose: Parallel complex event processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 191–200, New York, NY, USA, 2012. ACM.

[HJHF14] Thomas Heinze, Zbigniew Jerzak, Gregor Hackenbroich, and Christof Fetzer. Latency-aware elastic scaling for distributed data stream processing systems. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 13–22, New York, NY, USA, 2014. ACM.

[HJP+13] Thomas Heinze, Yuanzhen Ji, Yinying Pan, Franz Josef Grueneberger, Zbigniew Jerzak, and Christof Fetzer. Elastic complex event processing under varying query load. In *First International Workshop on Big Dynamic Distributed Data (BD3)*, 2013.

[HLR+13] Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. Mobile fog: A programming model for large-scale applications on the internet of things. In *Proceedings*

of the second ACM SIGCOMM workshop on Mobile cloud computing, pages 15–20. ACM, 2013.

[HPJF14] Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. Auto-scaling techniques for elastic data stream processing. In *2014 IEEE 30th International Conference on Data Engineering Workshops (ICDEW)*, pages 296–302. IEEE, 2014.

[HRM+15] Thomas Heinze, Lars Roediger, Andreas Meister, Yuanzhen Ji, Zbigniew Jerzak, and Christof Fetzer. Online parameter optimization for elastic data stream processing. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 276–287, New York, NY, USA, 2015. ACM.

[HSS+14] Martin Hirzel, Robert Soulé, Scott Schneider, Buğra Gedik, and Robert Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys*, 46(4):46:1–46:34, March 2014.

[IBM17] IBM. IBM Streams. `https://www.ibm.com/analytics/us/en/technology/stream-computing/`, 2017. [Online; accessed 11-Oct-2017].

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[Int14] International Data Corporation (IDC). The digital universe of opportunities: Rich data and the increasing value of the internet of things. `http://www.emc.com/leadership/digital-universe/2014iview/internet-of-things.htm`, April 2014. [Online; accessed 04-Jan-2018].

[JAA+06] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. Design, implementation, and evaluation of the linear road benchmark on the stream processing core. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 431–442, New York, NY, USA, 2006. ACM.

[JSGAW09] Gabriela Jacques-Silva, Bugra Gedik, Henrique Andrade, and Kun-Lung Wu. Language Level Checkpointing Support for Stream Processing Applications. In *Proceedings of the 39th IEEE/IFIP International Conference on Dependable Systems and Networks, DSN '09*, pages 145–154, June 2009.

[JZ14] Zbigniew Jerzak and Holger Ziekow. The debs 2014 grand challenge. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 266–269, New York, NY, USA, 2014. ACM.

[Kal14] Nathan Kallus. Predicting crowd behavior with big public data. In *Proceedings of the 23rd International Conference on World Wide Web*, WWW '14 Companion, pages 625–630, New York, NY, USA, 2014. ACM.

[KBF+15] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, Karthik Ramasamy, and Siddarth Taneja. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 239–250, New York, NY, USA, 2015. ACM.

[KBG08] YongChul Kwon, Magdalena Balazinska, and Albert Greenberg. Fault-tolerant Stream Processing using a Distributed, Replicated File System. *Proceedings of VLDB Endow.*, pages 574–585, 2008.

[KCFP12] Evangelia Kalyvianaki, Themistoklis Charalambous, Marco Fiscato, and Peter Pietzuch. Overload management in data stream processing systems with latency guarantees. In *7th IEEE International Workshop on Feedback Computing (Feedback Computing'12)*, 2012.

[KKR10] Gerald G. Koch, Boris Koldehofe, and Kurt Rothermel. Cordies: Expressive event correlation in distributed systems. In *Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems*, DEBS '10, pages 26–37, New York, NY, USA, 2010. ACM.

[KLB08] Lukas Kencl and Jean-Yves Le Boudec. Adaptive load sharing for network processors. *IEEE/ACM Trans. Netw.*, 16(2):293–306, April 2008.

[KMR+13] Boris Koldehofe, Ruben Mayer, Umakishore Ramachandran, Kurt Rothermel, and Marco Völz. Rollback-recovery without checkpoints in distributed event processing systems. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 27–38, New York, NY, USA, 2013. ACM.

[KORR12] Boris Koldehofe, Beate Ottenwälder, Kurt Rothermel, and Umakishore Ramachandran. Moving range queries in distributed complex event processing. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 201–212, New York, NY, USA, 2012. ACM.

[Kou06] Samuel Kounev. Performance modeling and evaluation of distributed component-based systems using queueing petri nets. *IEEE Transactions on Software Engineering*, 32(7):486–502, 2006.

[KSS15] Ilya Kolchinsky, Izchak Sharfman, and Assaf Schuster. Lazy evaluation methods for detecting complex events. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, pages 34–45, New York, NY, USA, 2015. ACM.

[KWF+16] Alexandros Koliousis, Matthias Weidlich, Raul Castro Fernandez, Alexander L. Wolf, Paolo Costa, and Peter Pietzuch. Saber: Window-based hybrid stream processing for heterogeneous architectures. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 555–569, New York, NY, USA, 2016. ACM.

[LDGB13] Katrina LaCurts, Shuo Deng, Ameesh Goyal, and Hari Balakrishnan. Choreo: Network-aware task placement for cloud applications. In *Proceedings of the 2013 Internet Measurement Conference*, IMC '13, pages 191–204, 2013.

[LFV+12] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. The vertica analytic database: C-store 7 years later. *Proc. VLDB Endow.*, 5(12):1790–1801, August 2012.

[LJK15] Björn Lohrmann, Peter Janacik, and Odej Kao. Elastic stream processing with latency guarantees. In *2015 IEEE 35th International Conference on Distributed Computing Systems*, pages 399–410, June 2015.

[LMT+05] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Rec.*, 34(1):39–44, March 2005.

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

[May12] Ruben Mayer. Wiederherstellung von Ereignisströmen in CEP-Systemen. Diploma thesis, University of Stuttgart, 2012.

[MBF11] André Martin, Andrey Brito, and Christof Fetzer. Active replication at (almost) no cost. In *SRDS '11: Proceedings of the 2011 30th IEEE International Symposium on Reliable Distributed Systems*, pages 21 –30, Washington, DC, USA, Oct 2011. IEEE Computer Society.

[MBF14] André Martin, Andrey Brito, and Christof Fetzer. Scalable and elastic realtime click stream analysis using streammine3g. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 198–205, New York, NY, USA, 2014. ACM.

[Men16] Gabriele Mencagli. A game-theoretic approach for elastic distributed data stream processing. *ACM Trans. Auton. Adapt. Syst.*, 11(2):13:1–13:34, June 2016.

[MGG+17] Ruben Mayer, Leon Graser, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. EmuFog: extensible and scalable emulation of Large-Scale fog computing infrastructures. In *2017 IEEE Fog World Congress (FWC) (FWC 2017)*, pages 103–108, Santa Clara, USA, October 2017.

[MGSR17a] Ruben Mayer, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. The fog makes sense: Enabling social sensing services with limited internet connectivity. In *Proceedings of the 2Nd International Workshop on Social Sensing*, SocialSens'17, pages 61–66, New York, NY, USA, 2017. ACM.

[MGSR17b] Ruben Mayer, Harshit Gupta, Enrique Saurez, and Umakishore Ramachandran. FogStore: toward a distributed data store for fog computing. In *2017 IEEE Fog World Congress (FWC) (FWC 2017)*, pages 128–133, Santa Clara, USA, October 2017.

[MKR14] Ruben Mayer, Boris Koldehofe, and Kurt Rothermel. Meeting predictable buffer limits in the parallel execution of event processing operators. In *Proceedings of the IEEE International Conference on Big Data*, BigData '14, pages 402–411. IEEE, 2014.

[MKR15] Ruben Mayer, Boris Koldehofe, and Kurt Rothermel. Predictable low-latency event detection with parallel complex event processing. *Internet of Things Journal, IEEE*, 2(4):274–286, Aug 2015.

[MM13] Ruixue Mao and Guoqiang Mao. Road traffic density estimation in vehicular networks. In *Proceedings of the 2013 IEEE Wireless Communications and Networking Conference*, WCNC '13, pages 4653–4658, April 2013.

[MMA17] Christian Mayer, Ruben Mayer, and Majd Abdo. Streamlearner: Distributed incremental machine learning on event streams: Grand challenge. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, DEBS '17, pages 298–303, New York, NY, USA, 2017. ACM.

[MMTR16] Ruben Mayer, Christian Mayer, Muhammad Adnan Tariq, and Kurt Rothermel. Graphcep: Real-time data analytics using parallel complex event and graph processing. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, pages 309–316, New York, NY, USA, 2016. ACM.

[MP14] Christopher Mutschler and Michael Philippsen. Adaptive speculative processing of out-of-order event streams. *ACM Trans. Internet Technol.*, 14(1):4:1–4:24, August 2014.

[MSD+15] André Martin, Tiaraju Smaneoto, Tobias Dietze, Andrey Brito, and Christof Fetzer. User-constraint and self-adaptive fault tolerance for event stream processing systems. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 462–473, June 2015.

[MSPC12] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497 – 1516, 2012.

[MST+17] Ruben Mayer, Ahmad Slo, Muhammad Adnan Tariq, Kurt Rothermel, Manuel Gräber, and Umakishore Ramachandran. Spectre: Supporting consumption policies in window-based parallel complex event processing. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference*, Middleware '17, pages 161–173, New York, NY, USA, 2017. ACM.

[MTR16] Ruben Mayer, Muhammad Adnan Tariq, and Kurt Rothermel. Real-time Batch Scheduling in Data-Parallel Complex Event Processing. Technical Report 2016/04, University of Stuttgart, 2016.

[MTR17] Ruben Mayer, Muhammad Adnan Tariq, and Kurt Rothermel. Minimizing communication overhead in window-based parallel complex event processing. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, DEBS '17, pages 54–65, New York, NY, USA, 2017. ACM.

[OKR+14a] Beate Ottenwälder, Boris Koldehofe, Kurt Rothermel, Kirak Hong, David Lillethun, and Umakishore Ramachandran. Mcep: A mobility-aware complex event processing system. *ACM Trans. Internet Technol.*, 14(1):6:1–6:24, August 2014.

[OKR+14b] Beate Ottenwälder, Boris Koldehofe, Kurt Rothermel, Kirak Hong, and Umakishore Ramachandran. Recep: Selection-based reuse for distributed complex event processing. In *Proceedings of the 8th ACM International Conference on Distributed Event-Based Systems*, DEBS '14, pages 59–70, New York, NY, USA, 2014. ACM.

[OKRR13] Beate Ottenwälder, Boris Koldehofe, Kurt Rothermel, and Umakishore Ramachandran. Migcep: Operator migration for mobility driven distributed complex event processing. In *Proceedings of the 7th ACM International Conference on Distributed Event-based Systems*, DEBS '13, pages 183–194, New York, NY, USA, 2013. ACM.

[OMK14] Beate Ottenwälder, Ruben Mayer, and Boris Koldehofe. Distributed complex event processing for mobile large-scale video applications.

In *Proceedings of the Posters & Demos Session*, Middleware Posters and Demos '14, pages 5–6, New York, NY, USA, 2014. ACM.

[Ora17] Oracle. Oracle CEP. `http://www.oracle.com/technetwork/middleware/complex-event-processing/overview/index.html`, 2017. [Online; accessed 11-Oct-2017].

[Ott16] Beate Ottenwälder. *Mobility-awareness in complex event processing systems*. PhD thesis, University of Stuttgart, 2016.

[PLS+06] Peter Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo Seltzer. Network-aware operator placement for stream-processing systems. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 49–49, April 2006.

[RBQ16] Nicoló Rivetti, Yann Busnel, and Leonardo Querzoni. Load-aware shedding in stream processing systems. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, pages 61–68, New York, NY, USA, 2016. ACM.

[RDR10] Stamatia Rizou, Frank Dürr, and Kurt Rothermel. Solving the multi-operator placement problem in large-scale operator networks. In *Computer Communications and Networks (ICCCN), 2010 Proceedings of 19th International Conference on*, pages 1–6, 2010.

[RLTB10] Martin Randles, David Lamb, and Azzelarabe Taleb-Bendiab. A comparative study into distributed load balancing algorithms for cloud computing. In *2010 IEEE 24th International Conference on Advanced Information Networking and Applications Workshops*, pages 551–556, April 2010.

[SAG+09] Scott Schneider, Henrique Andrade, Bugra Gedik, Alain Biem, and Kung-Lung Wu. Elastic scaling of data parallel operators in stream processing. In *2009 IEEE International Symposium on Parallel Distributed Processing*, pages 1–12, May 2009.

[Sch90] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[Sch15] Björn Schilling. *Efficient and secure event correlation in heterogeneous environments*. PhD thesis, University of Stuttgart, 2015.

[SGLN+11] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera, and Vishaka Nanayakkara. Siddhi: A second look at complex event processing architectures. In *Proceedings of the 2011 ACM Workshop on Gateway Computing Environments*, GCE '11, pages 43–50, New York, NY, USA, 2011. ACM.

[SGRM17] Enrique Saurez, Harshit Gupta, Umakishore Ramachandran, and Ruben Mayer. Demo abstract: Fog computing for improving user application interaction and context awareness. In *2017 IEEE/ACM Second International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 281–282, April 2017.

[SHGW12] Scott Schneider, Martin Hirzel, Bugra Gedik, and Kun-Lung Wu. Auto-parallelizing stateful distributed streaming applications. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 53–64, New York, NY, USA, 2012. ACM.

[SHL+16] Enrique Saurez, Kirak Hong, Dave Lillethun, Umakishore Ramachandran, and Beate Ottenwälder. Incremental deployment and migration of geo-distributed situation awareness applications in the fog. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, DEBS '16, pages 258–269, New York, NY, USA, 2016. ACM.

[SKR11] Björn Schilling, Boris Koldehofe, and Kurt Rothermel. Efficient and distributed rule placement in heavy constraint-driven event systems. In *2011 IEEE International Conference on High Performance Computing and Communications*, pages 355–364, Sept 2011.

[SKRR13] Björn Schilling, Boris Koldehofe, Kurt Rothermel, and Umakishore Ramachandran. Access policy consolidation for event processing systems. In *2013 Conference on Networked Systems*, pages 92–101, March 2013.

[SM11] Zoe Sebepou and Kostas Magoutis. Cec: Continuous eventual checkpointing for data stream processing operators. In *Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems and Networks, DSN '11*, pages 145 –156, 2011.

[SMMP09] Nicholas Poul Schultz-Møller, Matteo Migliavacca, and Peter Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 4:1–4:12, New York, NY, USA, 2009. ACM.

[Ste74] Michael A Stephens. Edf statistics for goodness of fit and some comparisons. *Journal of the American statistical Association*, 69(347):730–737, 1974.

[sto14] Storm, September 2014. http://storm-project.net/.

[SWG+15] Arik Senderovich, Matthias Weidlich, Avigdor Gal, Avishai Mandelbaum, Sarah Kadish, and Craig A. Bunnell. *Discovery and Validation of Queueing Networks in Scheduled Processes*, pages 417–433. Springer International Publishing, Cham, 2015.

[SWGM15] Arik Senderovich, Matthias Weidlich, Avigdor Gal, and Avishai Mandelbaum. Queue mining for delay prediction in multi-class service processes. *Information Systems*, 53(Supplement C):278 – 295, 2015.

[SZG10] Hendrik Schweppe, Armin Zimmermann, and Daniel Grill. Flexible on-board stream processing for automotive sensor data. *IEEE Transactions on Industrial Informatics*, 6(1):81–92, Feb 2010.

[TcZ+03] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 309–320. VLDB Endowment, 2003.

[TcZ07] Nesime Tatbul, Uğur Çetintemel, and Stan Zdonik. Staying fit: Efficient load shedding techniques for distributed stream processing. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 159–170. VLDB Endowment, 2007.

[THS17] Kanat Tangwongsan, Martin Hirzel, and Scott Schneider. Low-latency sliding-window aggregation in worst-case constant time. In *Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems*, DEBS '17, pages 66–77, New York, NY, USA, 2017. ACM.

[Tij06] Henk Tijms. New and old results for the m/d/c queue. *AEU - International Journal of Electronics and Communications*, 60(2):125 – 130, 2006.

[TLG16] Liang Tong, Yong Li, and Wei Gao. A hierarchical edge cloud architecture for mobile computing. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, April 2016.

[TZ06] Nesime Tatbul and Stan Zdonik. Window-aware load shedding for aggregation queries over data streams. In *Proceedings of the 32Nd International Conference on Very Large Data Bases*, VLDB '06, pages 799–810. VLDB Endowment, 2006.

[VBVB09] William Voorsluys, James Broberg, Srikumar Venugopal, and Rajkumar Buyya. Cost of virtual machine live migration in clouds: A performance evaluation. In *Cloud Computing. CloudCom 2009. 1st International Conference on*, pages 254–265. Springer Berlin/Heidelberg, 2009.

[VH08] András Varga and Rudolf Hornig. An overview of the omnet++ simulation environment. In *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools '08, pages 60:1–60:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[VKR11] Marco Völz, Boris Koldehofe, and Kurt Rothermel. Supporting strong reliability for distributed complex event processing systems. In *2011 IEEE International Conference on High Performance Computing and Communications*, pages 477–486, Sept 2011.

[WCN+09] Benjamin Wester, James Cowling, Edmund B. Nightingale, Peter M. Chen, Jason Flinn, and Barbara Liskov. Tolerating latency in repli-

cated state machines through client speculation. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 245–260, Berkeley, CA, USA, 2009. USENIX Association.

[WDR06] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 407–418, New York, NY, USA, 2006. ACM.

[WS017] WS02. WSO2 Complex Event Processor. `https://wso2.com/products/complex-event-processor/`, 2017. [Online; accessed 11-Oct-2017].

[XWB+13] Wenlei Xie, Guozhang Wang, David Bindel, Alan Demers, and Johannes Gehrke. Fast iterative graph computation with block updates. *Proc. VLDB Endow.*, 6(14):2014–2025, September 2013.

[ZDL+12] Matei Zaharia, Tathagata Das, Haoyuan Li, Scott Shenker, and Ion Stoica. Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'12, Berkeley, CA, USA, 2012. USENIX Association.

[Zil96] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI magazine*, 17(3):73, 1996.

[ZR11] Erik Zeitler and Tore Risch. Massive scale-out of expensive continuous queries. *Proceedings of the VLDB Endowment*, 4(11), 2011.

[ZU99] Detlef Zimmer and Rainer Unland. On the semantics of complex events in active database management systems. In *Proceedings of the 15th International Conference on Data Engineering (ICDE)*, pages 392–399, Mar 1999.

[ZWC07] Fred Zemke, Andrew Witkowski, and Mitch Cherniak. Pattern matching in sequences of rows (11). `http://www.cs.ucla.edu/classes/spring08/cs240B/notes/row-pattern-recogniton-11.pdf`, 2007. [Online; accessed 04-Jan-2018].

Erklärung

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

(Ruben Daniel Mayer)